McGill

Computer Organization

Processor Implementation

ECSE 324

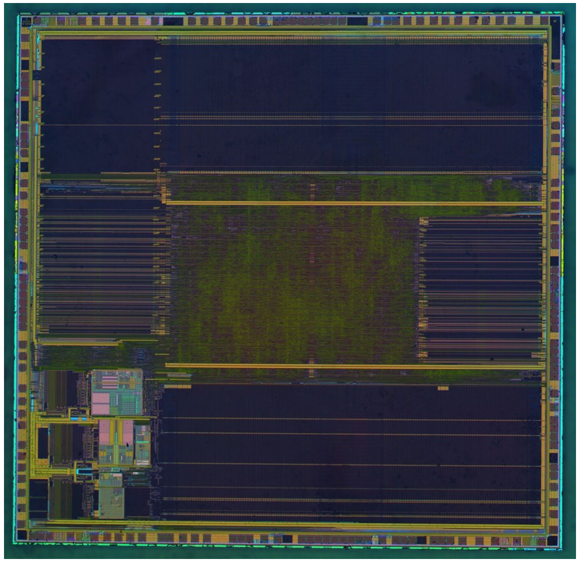
Fall 2020

Prof. Christophe Dubach

Original slides from Prof. Warren Gross. Some material from Hamacher, Vranesic, Zaky, and Manjikian, "Computer Organization and Embedded Systems, 6th ed", 2012, McGraw Hill and "Introduction to the ARM Processor using Altera Toolchain".

The Processor

- A processor is responsible for reading program instructions from the computer's memory and executing them.
- It *fetches* one instruction at a time.
- It *decodes* (interprets) the instruction.
- Then, it carries out the actions specified by the instruction



Die photo of an ARM Cortex-M3 processor

Processor building blocks

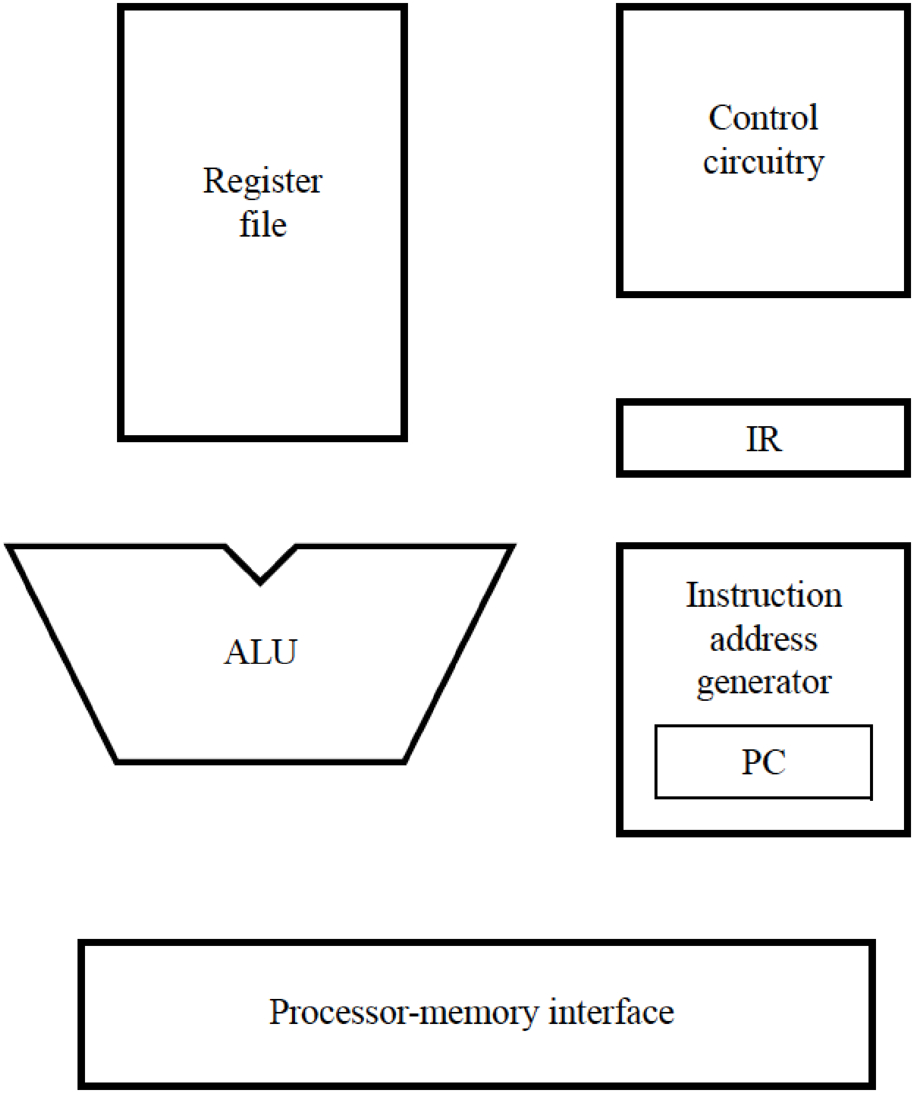
- `PC` holds the address of the next instruction to be fetched and executed
- Instruction is fetched into `IR`:

  `IR ⟵ [PC]`

- Instruction address generator updates `PC` (for straight line code):

  `PC ⟵ PC + 4`

- Control circuitry decodes the instruction and generates control signals that direct the datapath



Processor building blocks

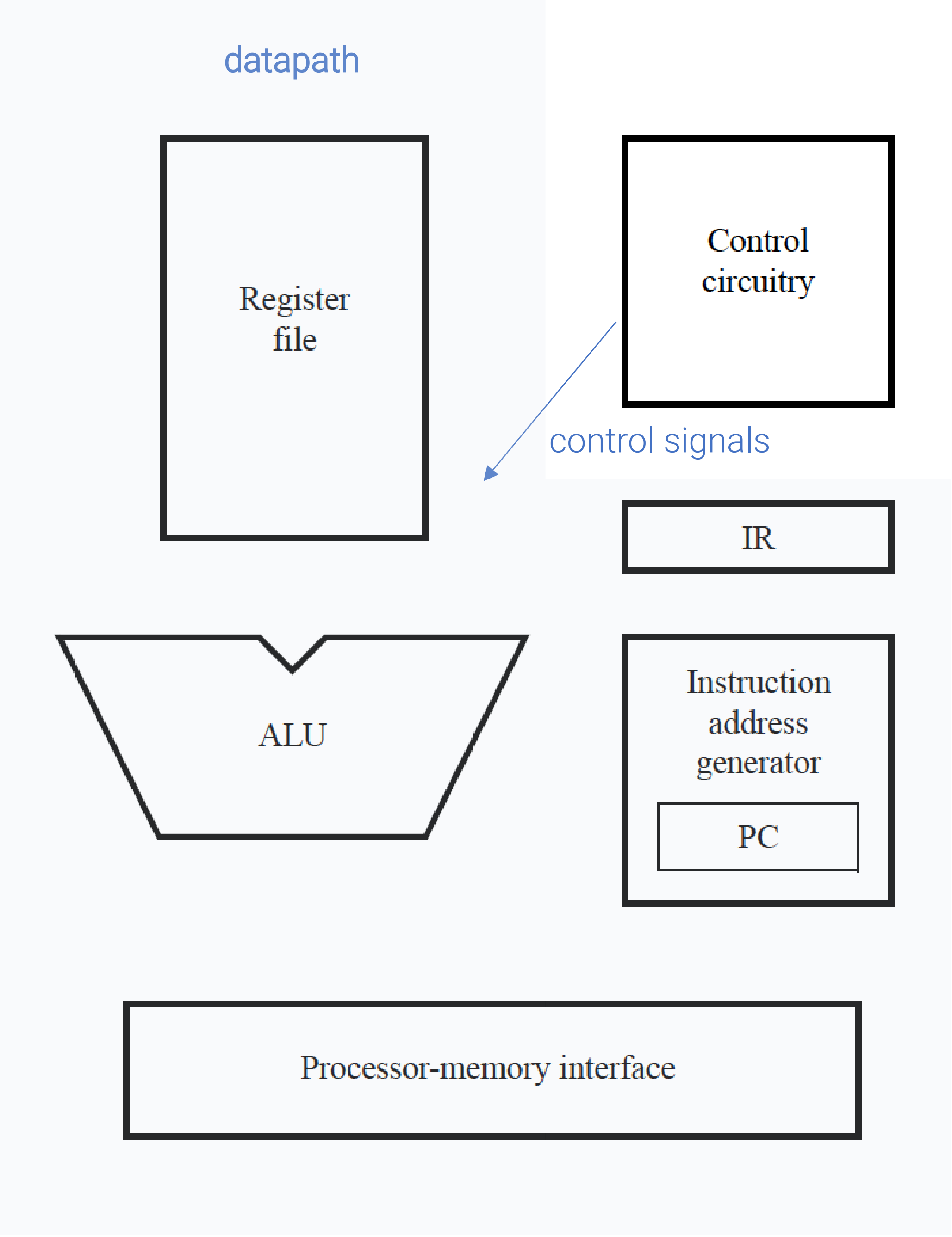
- `PC` holds the address of the next instruction to be fetched and executed
- Instruction is fetched into `IR`:

  `IR ⟵ [PC]`

- Instruction address generator updates `PC` (for straight line code):

  `PC ⟵ PC + 4`

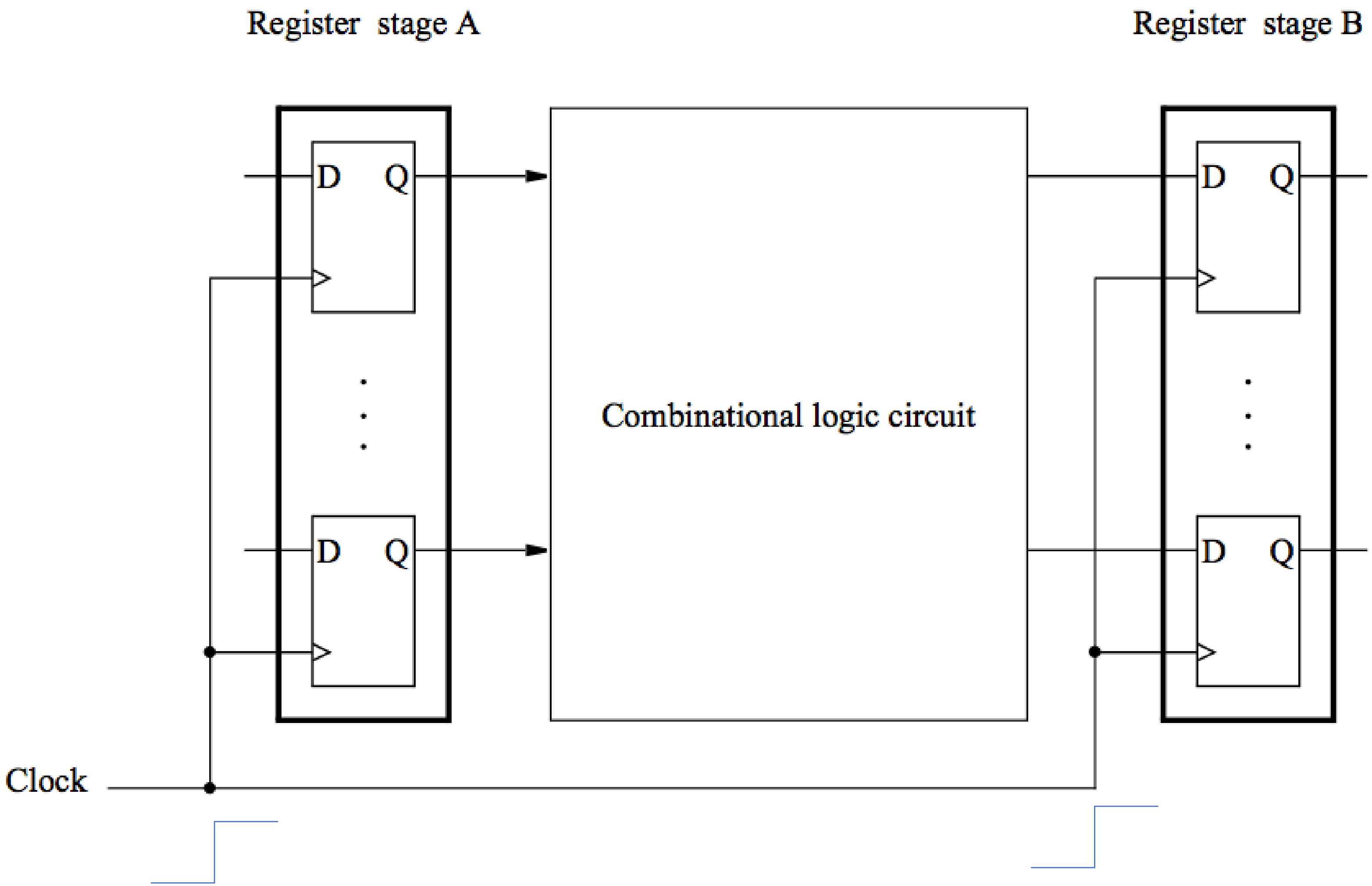
- Control circuitry decodes the instruction and generates control signals that direct the datapath



Datapath design

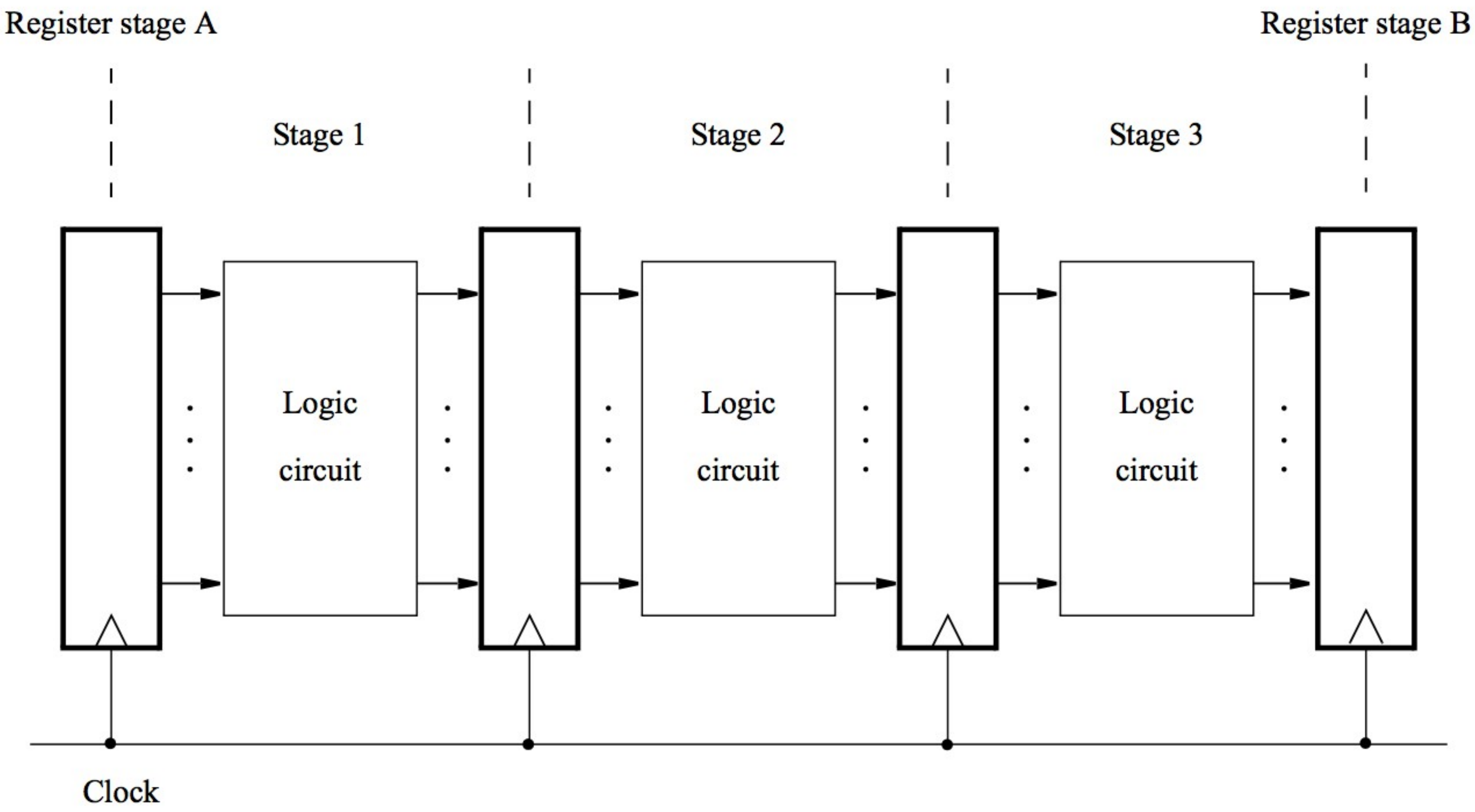
Textbook 5.1-5.4

- Contents of register A are processed and deposited in register B



clock period is determined by the delay of the combinational logic circuit

- Combinational circuits can be divided into simpler subcircuits that are cascaded into a multi-stage structure.
  - n stages: n clock cycles to complete the operation
  - Clock period can be shorter: 1/n
  - Can be pipelined (overlap the execution of several instructions)



Instruction execution

- In RISC machines, all instructions are executed in the same number of steps.
- Each step is carried out in a separate hardware stage.
- RISC processor design will be illustrated using five hardware stages
- The processor design will model the basic RISC instructions and addressing modes, but not every detail of an ARM ISA

Load instruction

`LDR R5, [R7, R8]` $R5 \longleftarrow [R7 + R8]$

offset mode

1. Fetch the instruction and increment the program counter

2. Decode the instruction and read the contents of registers `R7` and `R8` in the register file

3. Compute the effective address $Z \longleftarrow R7 + R8$

4. Read the memory source operand $[Z]$

5. Write the operand into the destination register

   $R5 \longleftarrow [Z]$

Arithmetic and logical instructions

`Add R3, R4, R5` $R3 \longleftarrow R4 + R5$

1. Fetch the instruction and increment the program counter
2. Decode the instruction and read registers `R4` and `R5` from the register file
3. Compute the sum $Z = R4 + R5$
4. No action why do we need this step ?
5. Write the result into the destination register

$$R3 \longleftarrow Z$$

Stage 4 (memory access) is not involved in this instruction.

Immediate Operands

`Add R3, R4, #1000` $\mathtt{R3} \longleftarrow \mathtt{R4} + \mathtt{1000}$

The immediate operand is given in the instruction word and can be found in the IR.

1. Fetch the instruction and increment the program counter
2. Decode the instruction and read register `R4` from the register file
3. Compute the sum $\mathtt{Z} = \mathtt{R4} + \mathtt{1000}$
4. No action
5. Write the result into the destination register

$$\mathtt{R3} \longleftarrow \mathtt{Z}$$

Load instruction (immediate)

`LDR R5, [R7, #X]` $R5 \longleftarrow [R7 + X]$

The immediate operand is given in the instruction word and can be found in the IR

1. Fetch the instruction and increment the program counter
2. Decode the instruction and read the contents of register `R7` in the register file
3. Compute the effective address $Z \longleftarrow R7 + X$
4. Read the memory source operand $[Z]$
5. Write the operand into the destination register

$$R5 \longleftarrow [Z]$$

Store instruction

`STR R6, [R8, #X]` `MEM[R8+X]` ⟵ `[R6]`

1. Fetch the instruction and increment the program counter

2. Decode the instruction and read the contents of registers `R6` and `R8` in the register file

3. Compute the effective address `Z = R8 + X`

4. Store the contents of register R6 into memory location `X + [R8]` `MEM[Z]` ⟵ `R6`

5. No action

Summary – Actions to implement an instruction

1. Fetch an instruction and increment the program counter.
2. Decode the instruction and read registers from the register file.
3. Perform an ALU operation.
4. Read or write memory data if the instruction involves a memory operand.
5. Write the result into the destination register

5 hardware stages will be needed

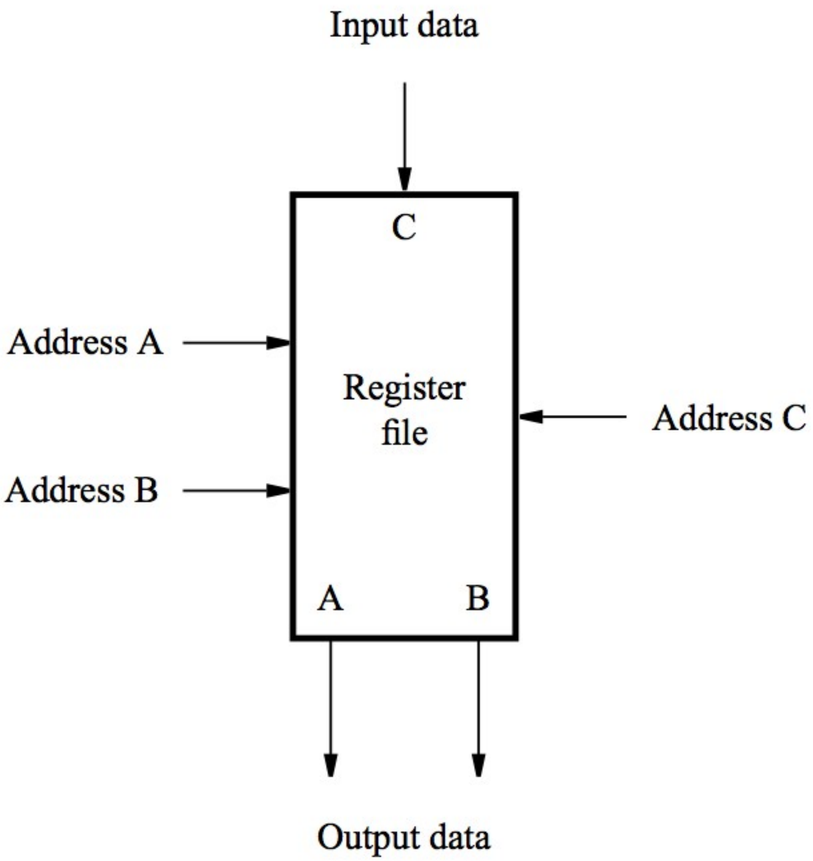
Stages 1, 2, and 3 will be used for all instructions

Stages 4 and 5 may not perform a useful action for some instructions

Hardware components: Register file

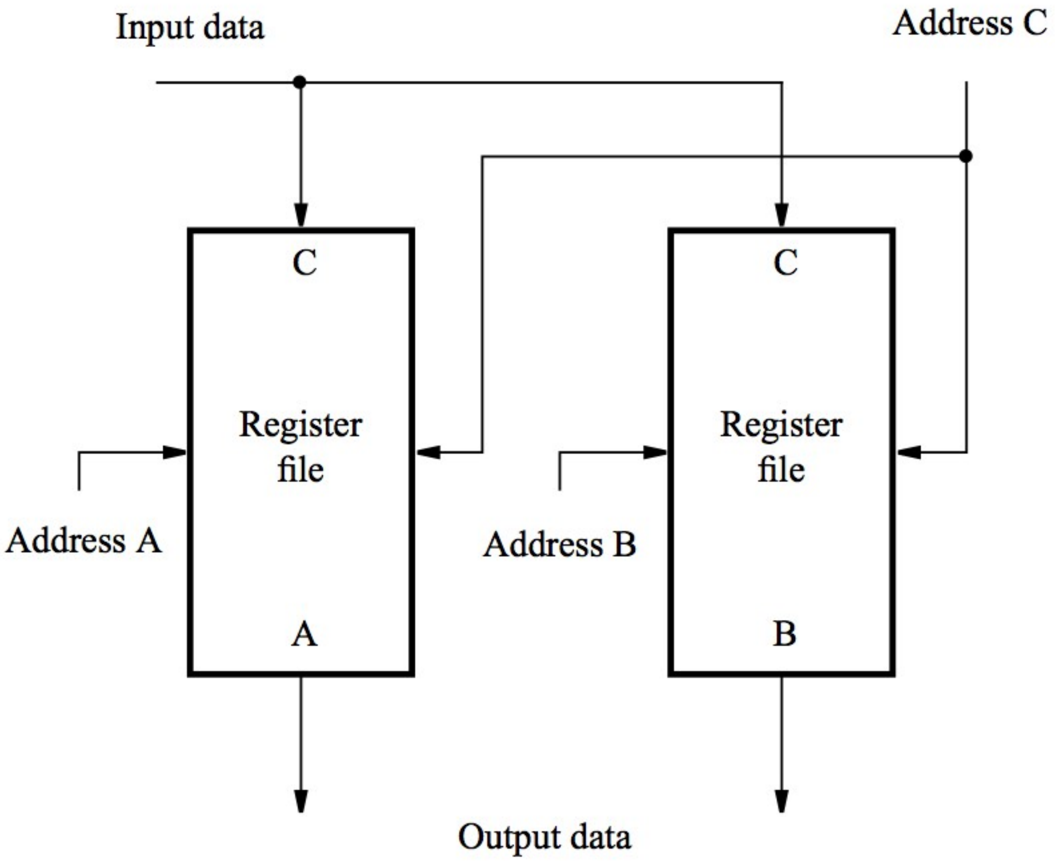
- A 2-port register file is needed to read the two source registers at the same time.

- It may be implemented using a 2-port memory.



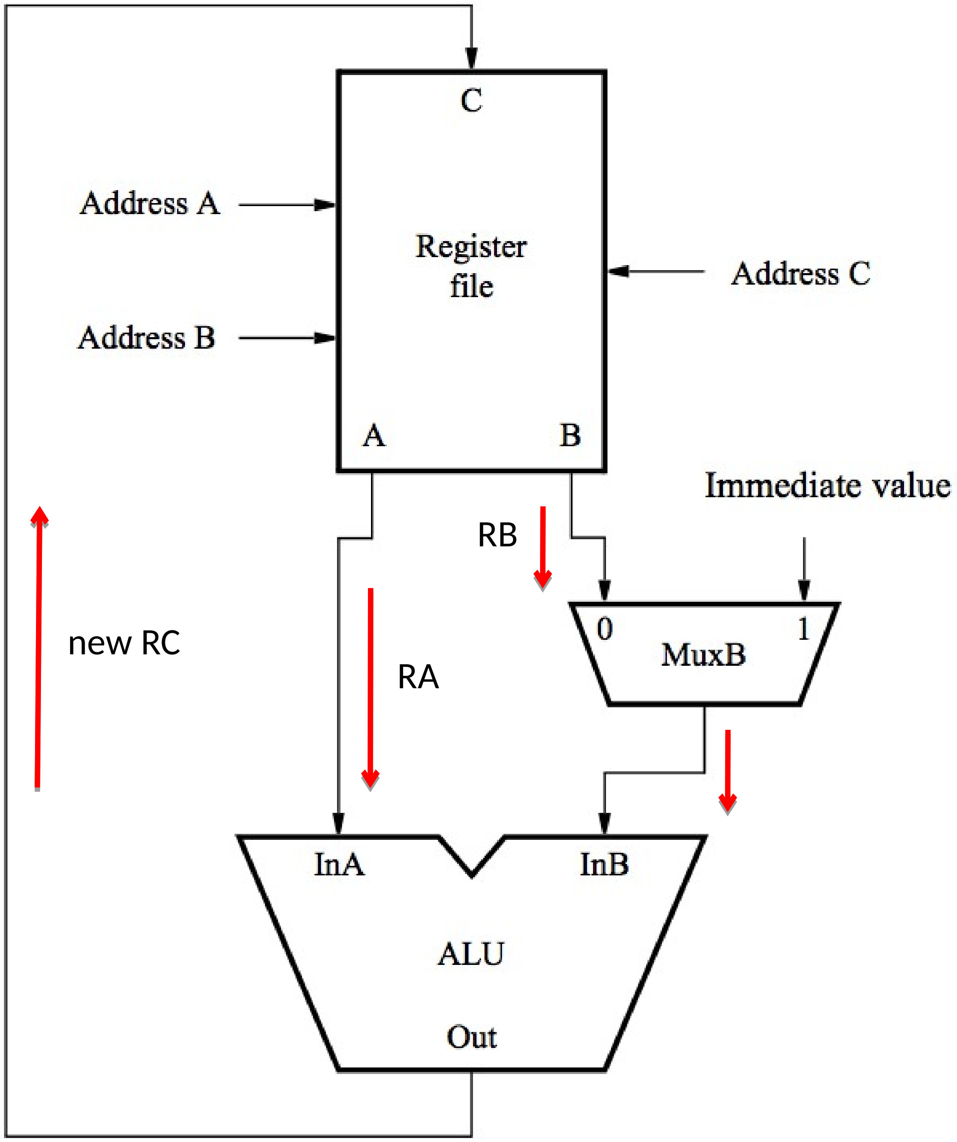
Alternative implementation of 2-port register file

- Using two single-ported memory blocks each containing a copy of the register file

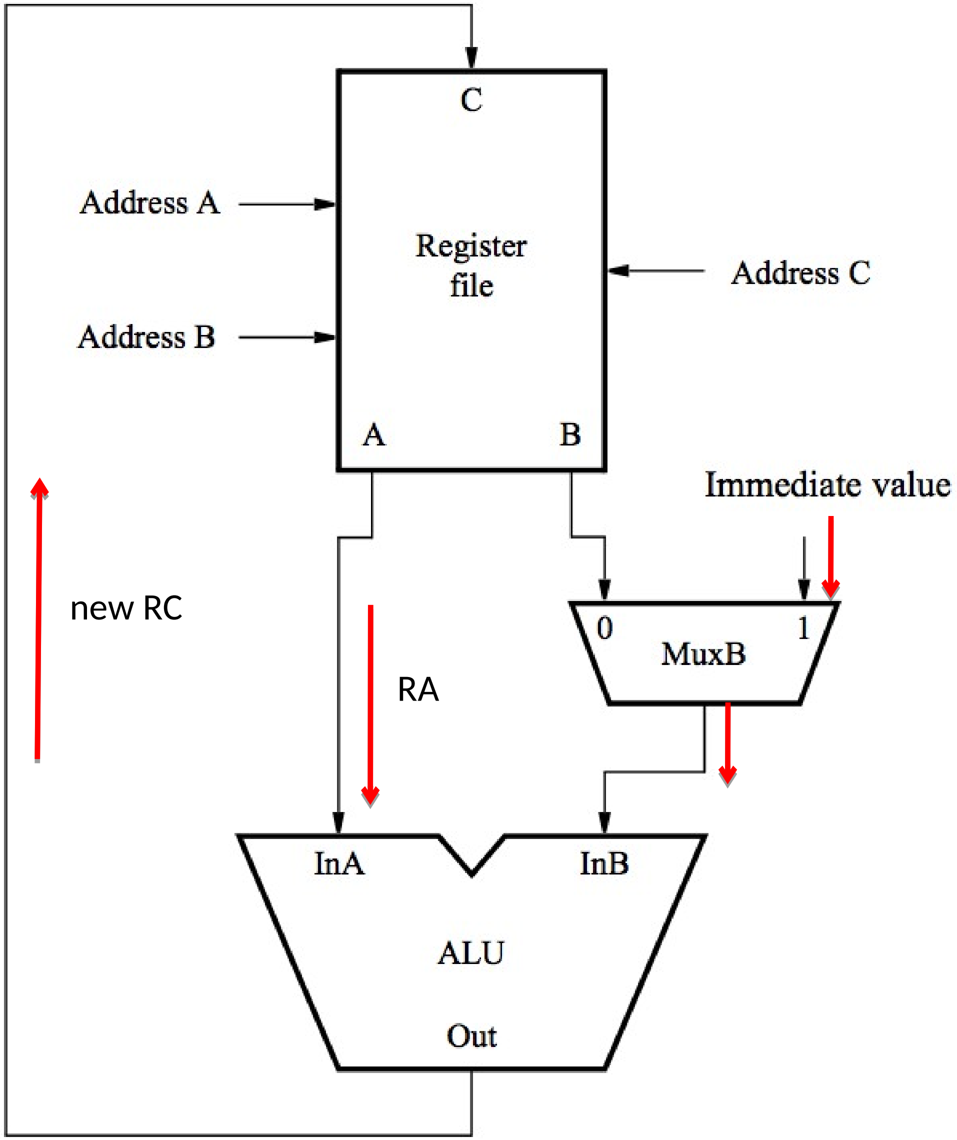


ALU

- Both source operands and the destination location are in the register file.
- Conceptual single-cycle view of an arithmetic or logical instruction with two source operands in registers

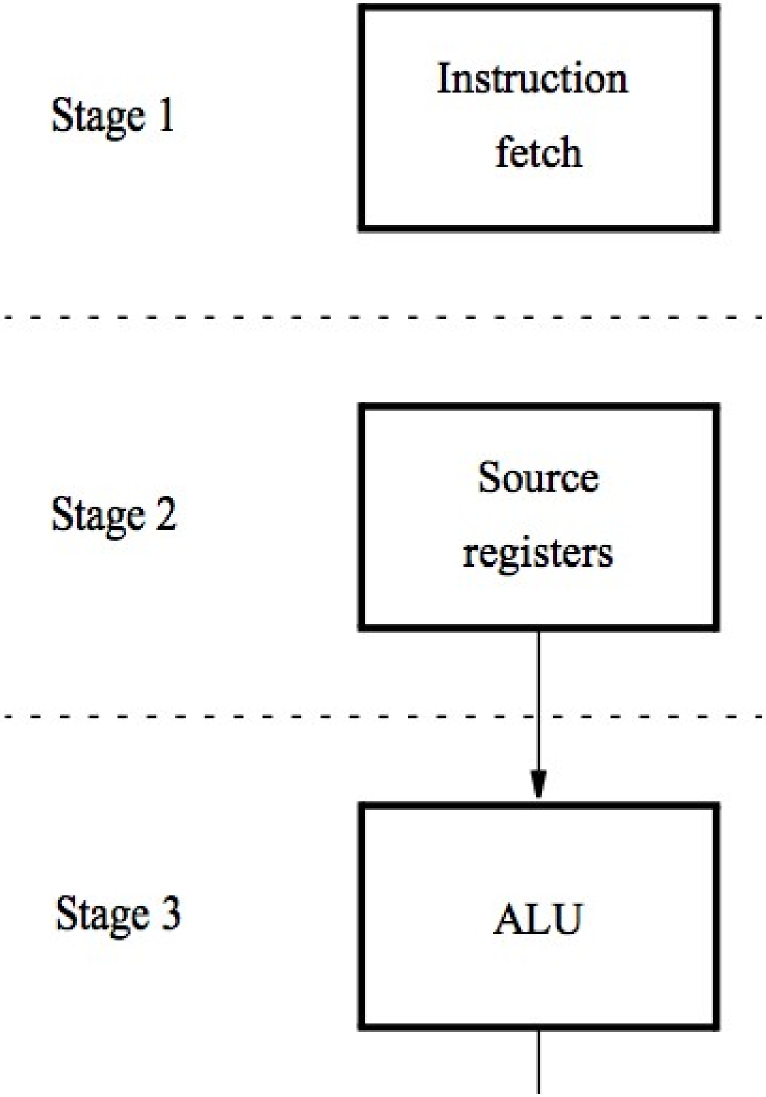


- One of the source operands is the immediate value in `IR`



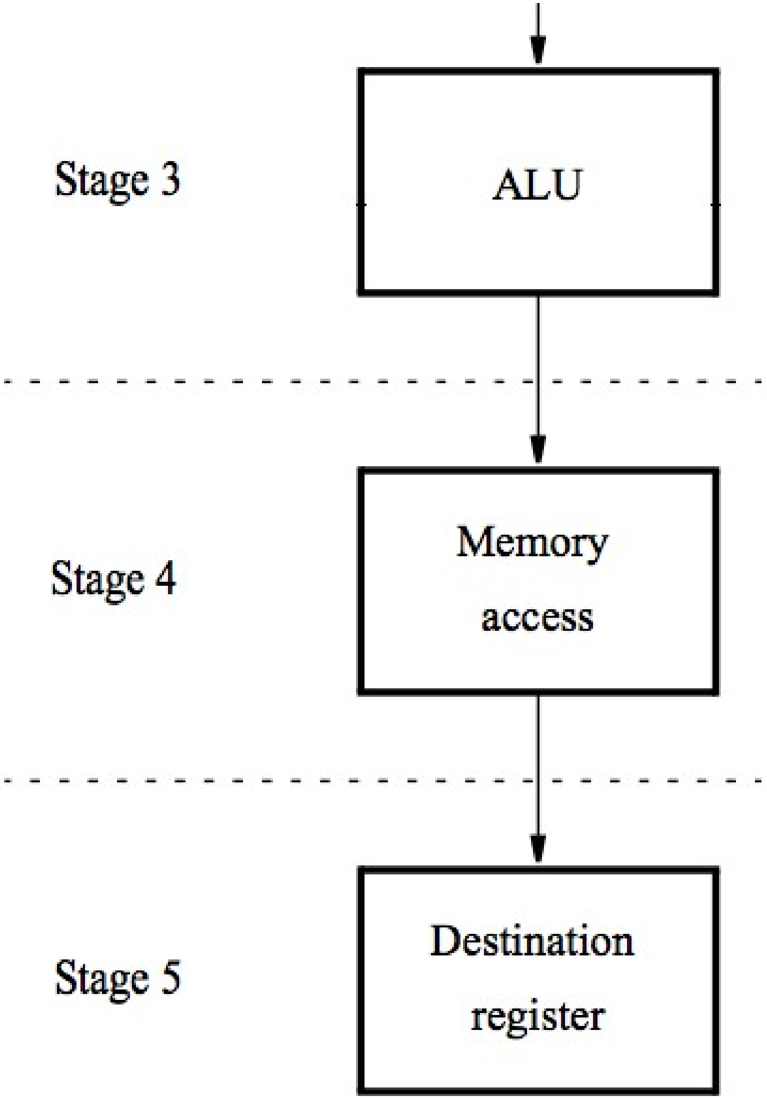
A 5-stage implementation of a RISC processor

- Instruction processing moves from stage to stage in every clock cycle, starting with fetch.
- The instruction is decoded and the source registers are read in stage 2.
- Computation takes place in the ALU in stage 3.



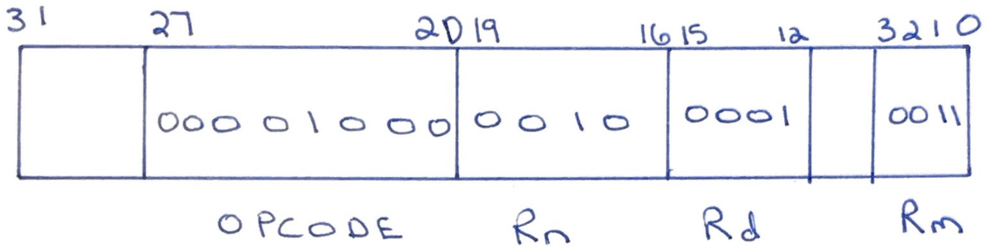
A 5-stage implementation of a RISC processor

- If a memory operation is involved, it takes place in stage 4.
- The result of the instruction is stored in the destination register in stage 5.

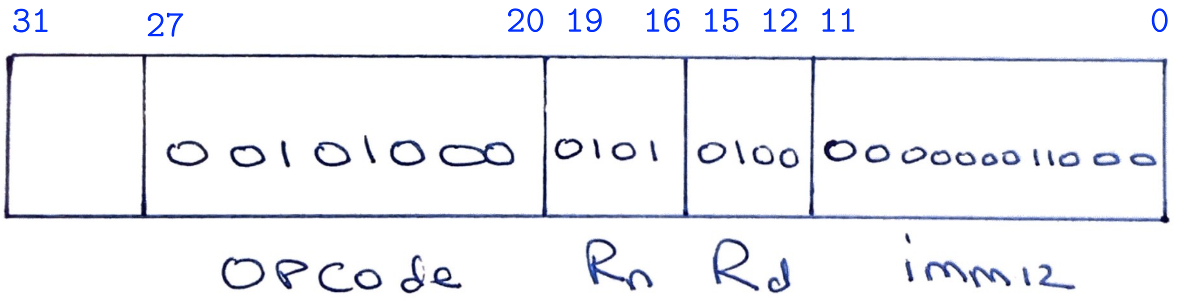


How can we decode the instruction and read the registers at the same time?

- In a RISC ISA, the register fields are always in the same positions in the instruction. If the registers were not needed by that instruction, the retrieved values will be ignored.



ADD R1, R3, R2



ADD R4, R5, #24

31 28	27 20	19 16	15 12	11 0
Condition	OP code	R*n*	R*d*	Offset or R*m*

Load/Store encoding

Waiting for memory

- We have assumed that all memory accesses take one clock cycle. Is this realistic?
    - Mostly true if we use a cache!
- In the case of cache miss, the processor must be stalled to wait for the memory access to complete (a variable number of cycles)
- The processor-memory interface generates a signal called *Memory Function Completed (**MFC**)*
- *Processor extends the duration of the memory step (in units of clock cycles) until MFC is asserted*

The datapath – Stages 2 to 5

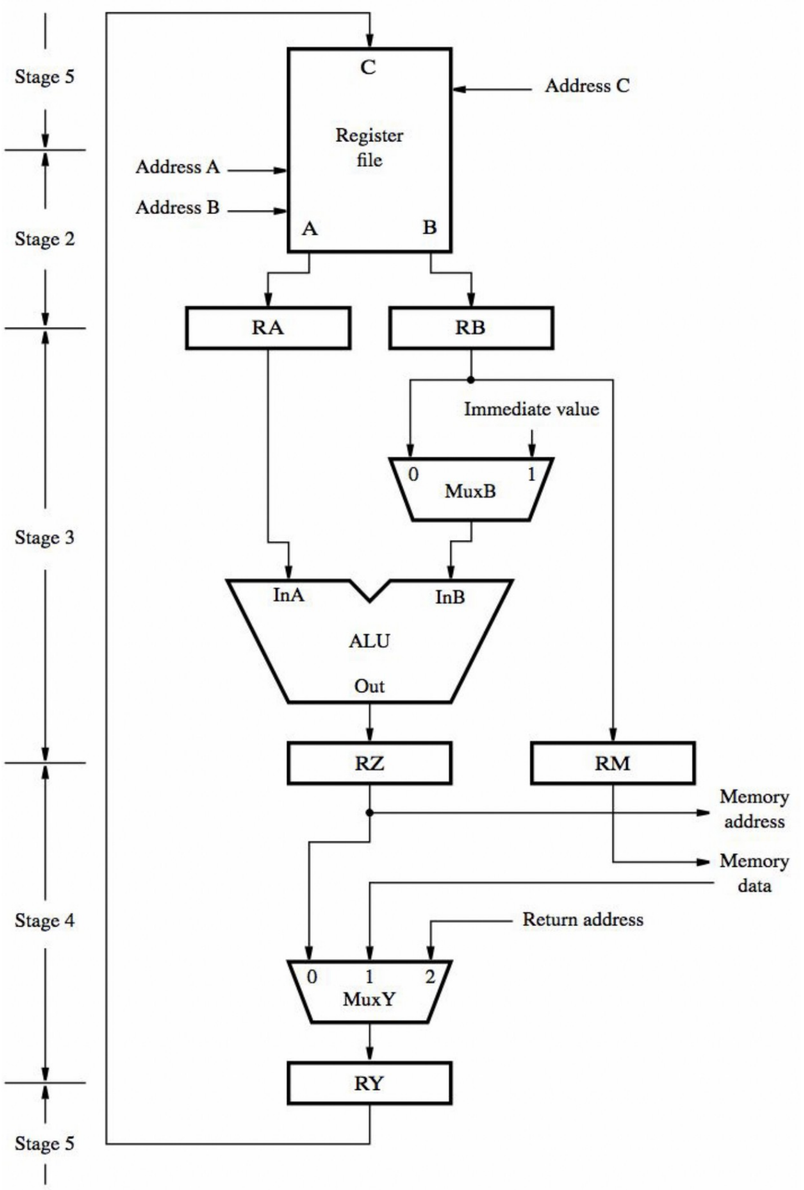
- Register file,
  used in stages 2 and 5

- Multicycle: Inter-stage registers `RA`, `RB`, `RZ`, `RY` needed to carry data from one stage to the next

- ALU stage

- Memory stage

- Final stage to store result to the register file

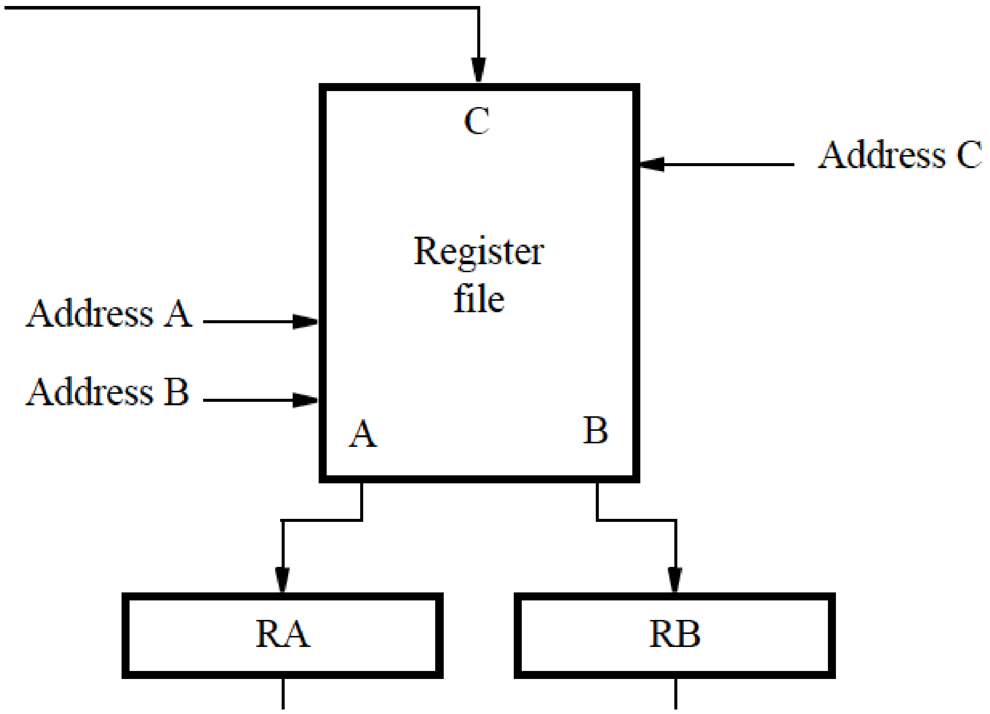


Register file – Stages 2 & 5

- Address inputs are connected to the corresponding fields in `IR`

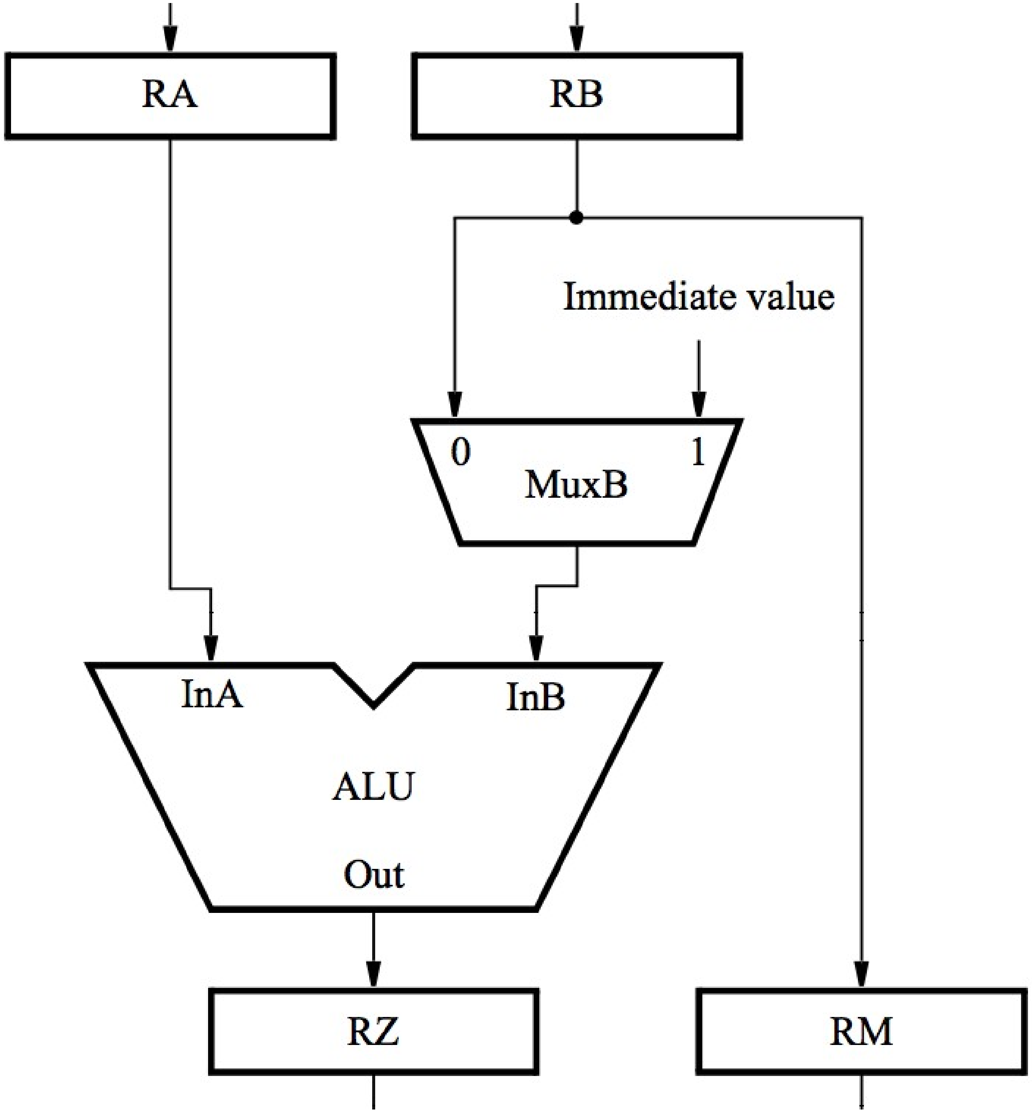
- Source registers are read in stage 2; their contents are stored in `RA` and `RB`

- In stage 5, the result of the instruction is stored in the destination register selected by `Address C`



ALU stage

- ALU performs calculation specified by the instruction.
- Multiplexer `MuxB` selects either `RB` or the Immediate field of `IR`
- Results stored in `RZ`
- Data to be written in the memory are transferred from `RB` to `RM`



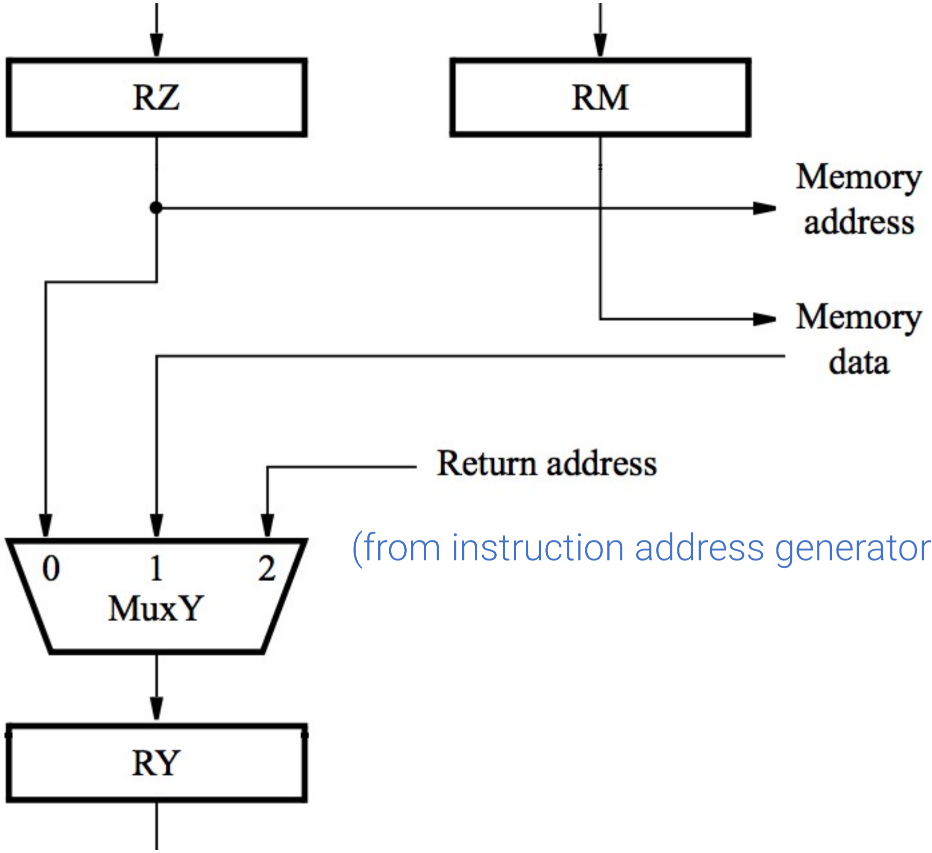
Memory stage

- For a memory instruction, `RZ` provides memory address, and `MuxY` selects read data to be placed in `RY`.

- `RM` provides data for a memory write operation.

- For a calculation instruction, `MuxY` selects `RZ` to be placed in `RY`.

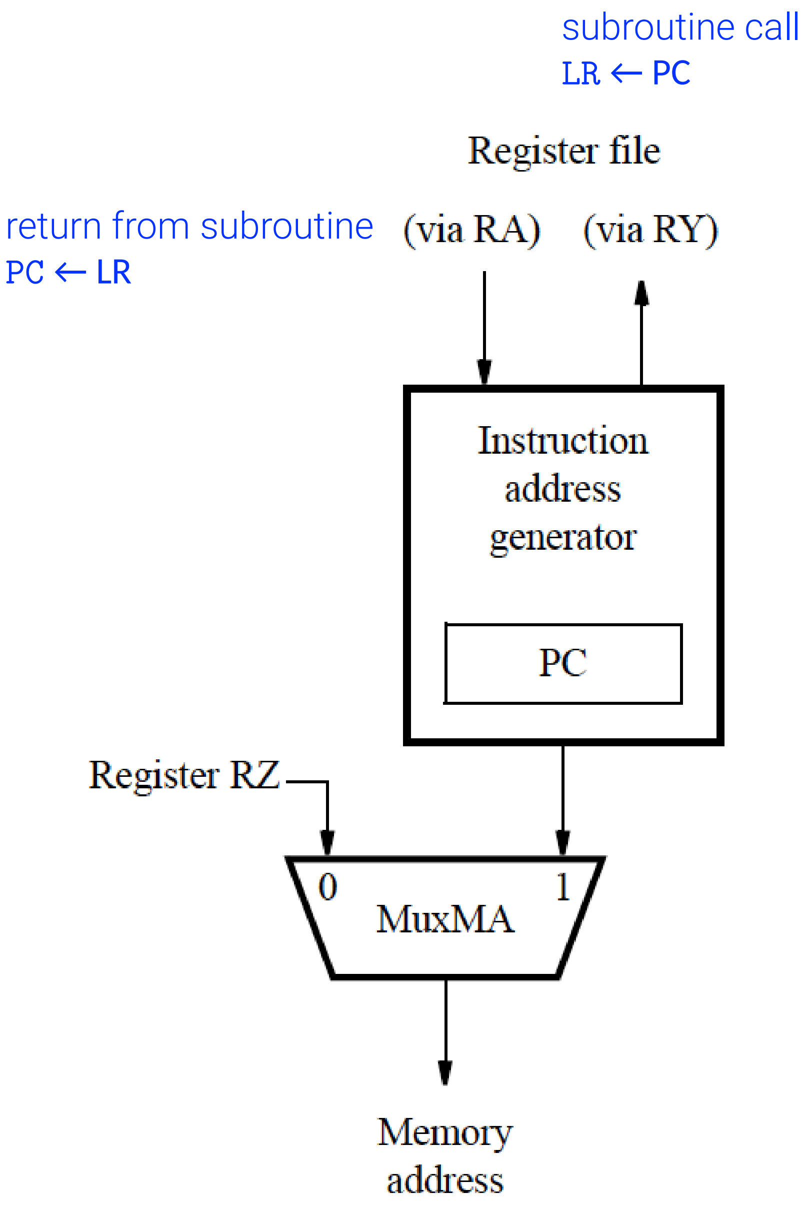
- Input 2 of `MuxY` is used in subroutine calls.



(from instruction address generator)

Memory address generation

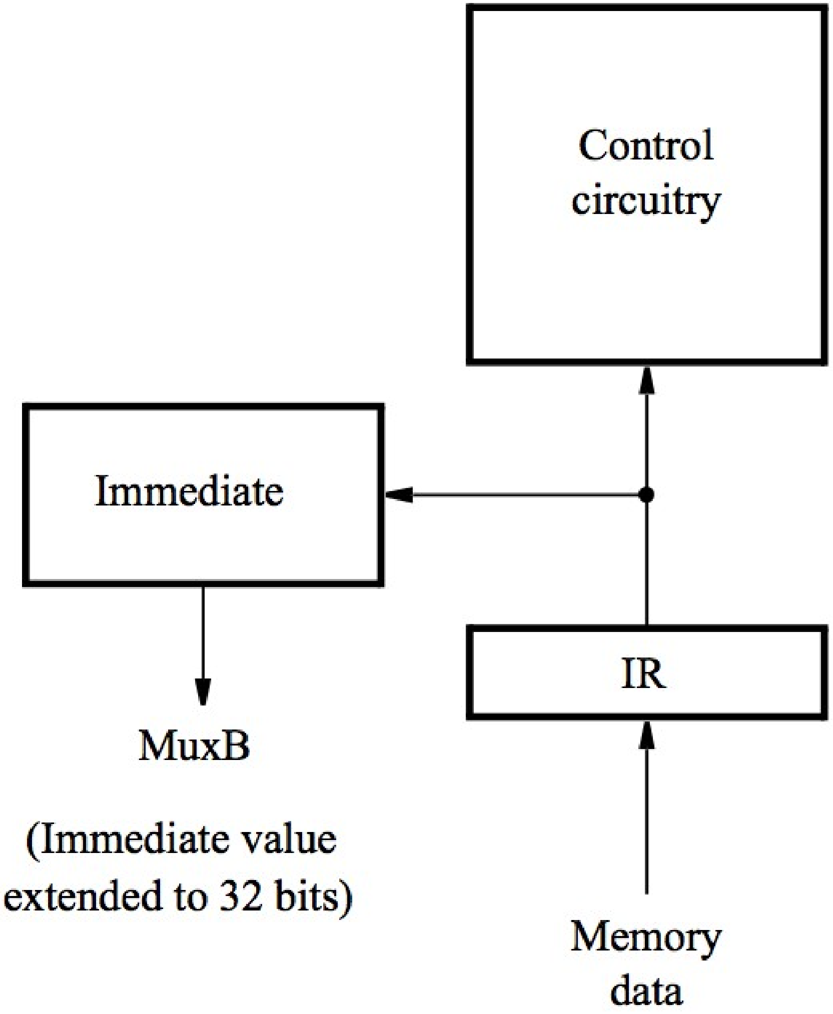
- `MuxMA` selects the `PC` when fetching instructions.
- The Instruction address generator increments the `PC` after fetching an instruction.
- It also generates branch and subroutine addresses.
- `MuxMA` selects `RZ` when reading/writing data operands.



Processor control section

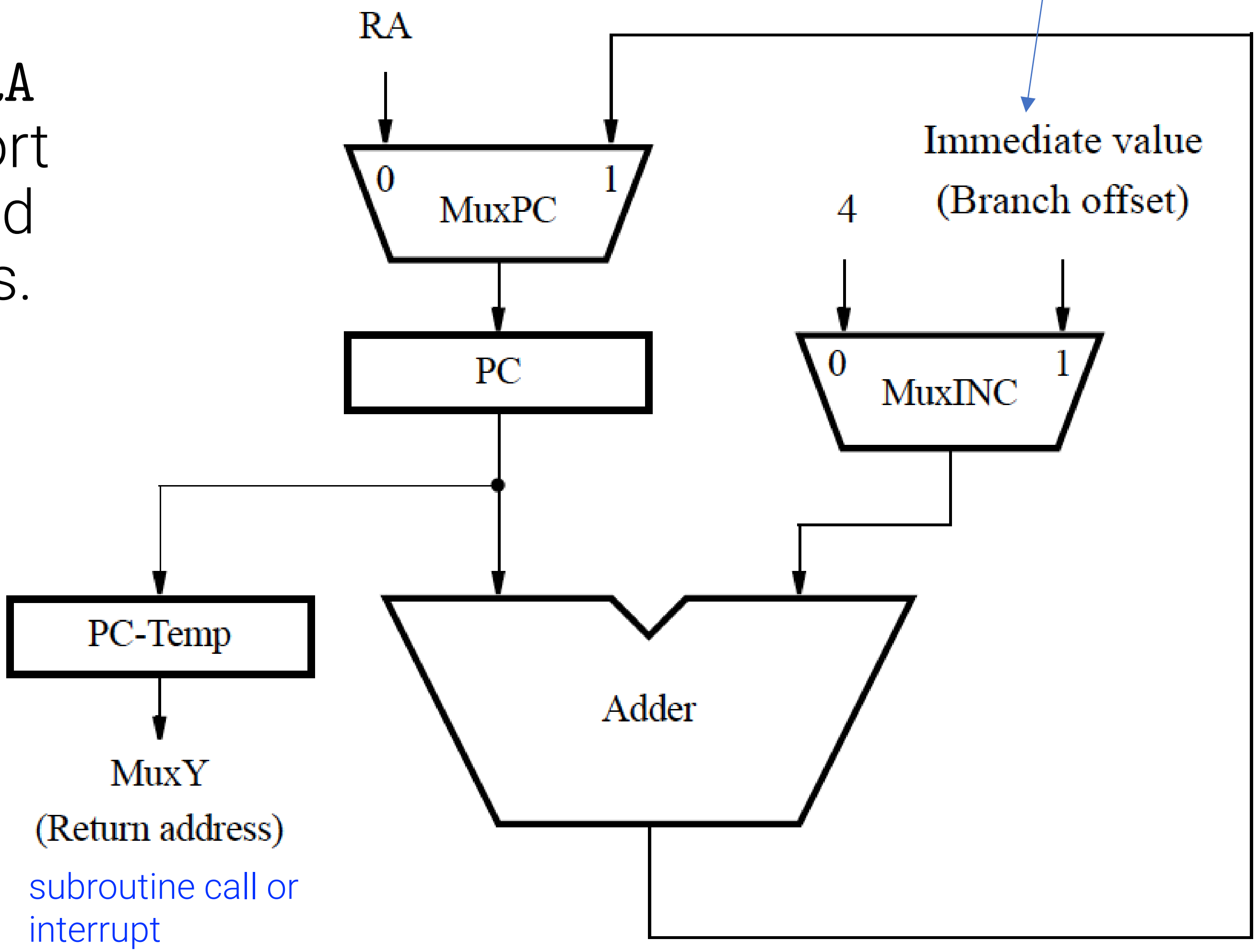
- When an instruction is read, it is placed in `IR`.
- The control circuitry decodes the instruction.
- It generates the control signals that drive all units.
- The Immediate block extends the immediate operand to 32 bits, according to the type of instruction

arithmetic: sign-extend
logic: zero padding



Instruction address generator

- Connections to registers `RY` and `RA` are used to support subroutine call and return instructions.



sign extended by immediate block on last slide

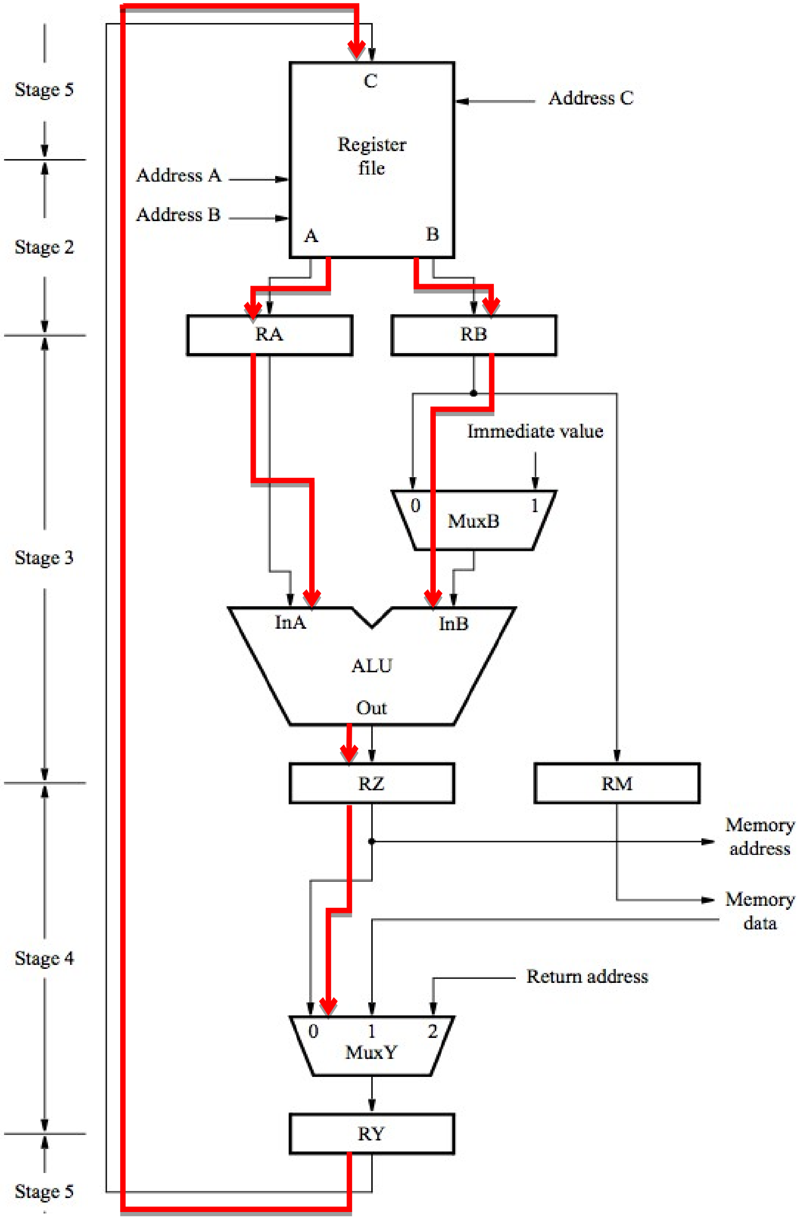
return from subroutine
PC ← LR

subroutine call or interrupt
LR ← PC

Example: `Add R3, R4, R5`

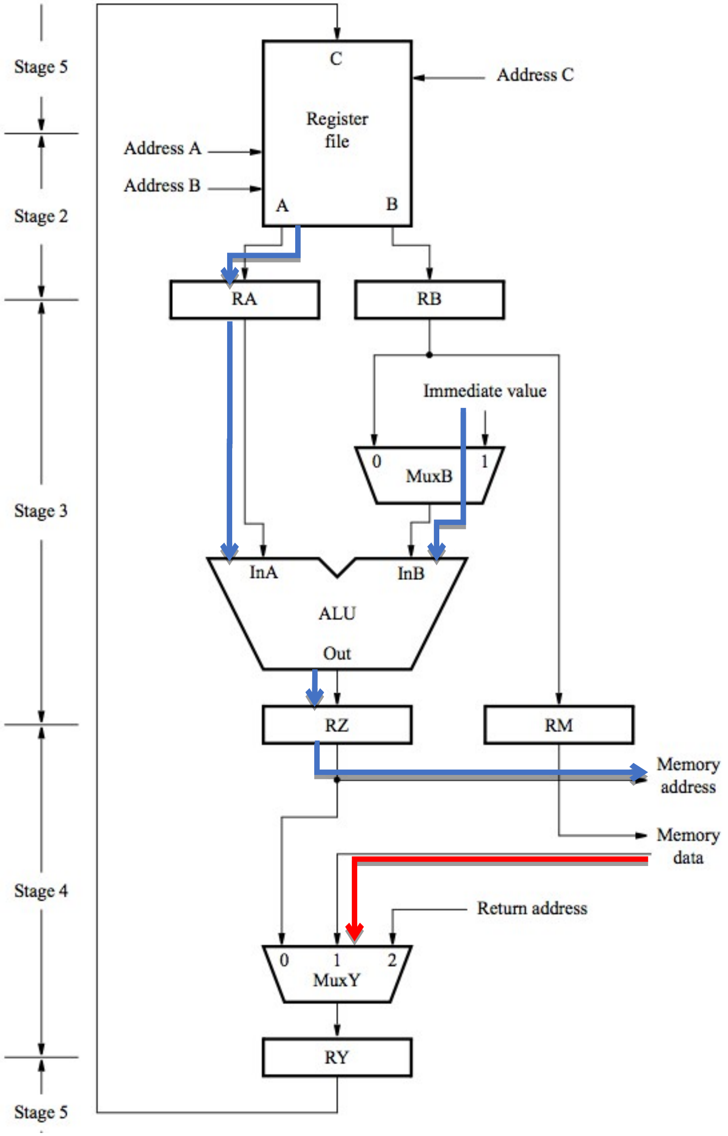
```
1.  Memory address ← PC,
    Read memory,
    IR ← Memory data,
    PC ← PC + 4

2.  Decode instruction,
    RA ← R4,
    RB ← R5

3.  RZ ← RA + RB

4.  RY ← RZ

5.  R3 ← RY
```

All of these actions happen in parallel



Example: LDR R5, [R7, #X]

```
1. Memory address ← PC,
   Read memory,
   IR ← Memory data,
   PC ← PC + 4

2. Decode instruction,
   RA ← R7

3. RZ ← RA + Immediate value X

4. Memory address ← RZ,
   Read memory,
   RY ← Memory data

5. R5 ← RY
```



Example: STR R6, [R8, #X]

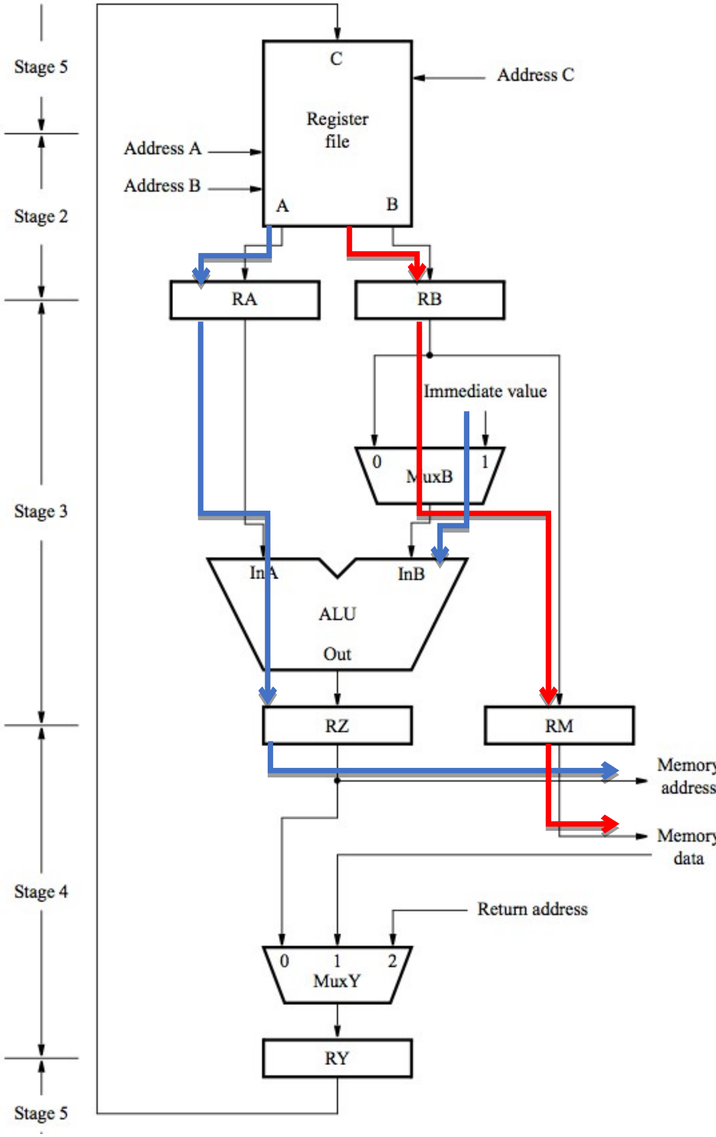
1. Memory address ← PC,
   Read memory,
   IR ← Memory data,
   PC ← PC + 4

2. Decode instruction,
   RA ← R8,
   RB ← R6

3. RZ ← RA + Immediate value X,
   RM ← RB

4. Memory address ← RZ,
   Memory data ← RM,
   Write memory

5. No action



Unconditional branch

```
1. Memory address ← PC,
   Read memory,
   IR ← Memory data,
   PC ← PC + 4

2. Decode instruction

3. PC ← PC + Branch offset

4. No action

5. No action
```

Conditional branch

- We will illustrate the generic RISC branch, that does not use condition codes.
- Instead, RISC branches compares values in registers and tests for some condition

e.g. `BEQ R5, R6, LOOP`

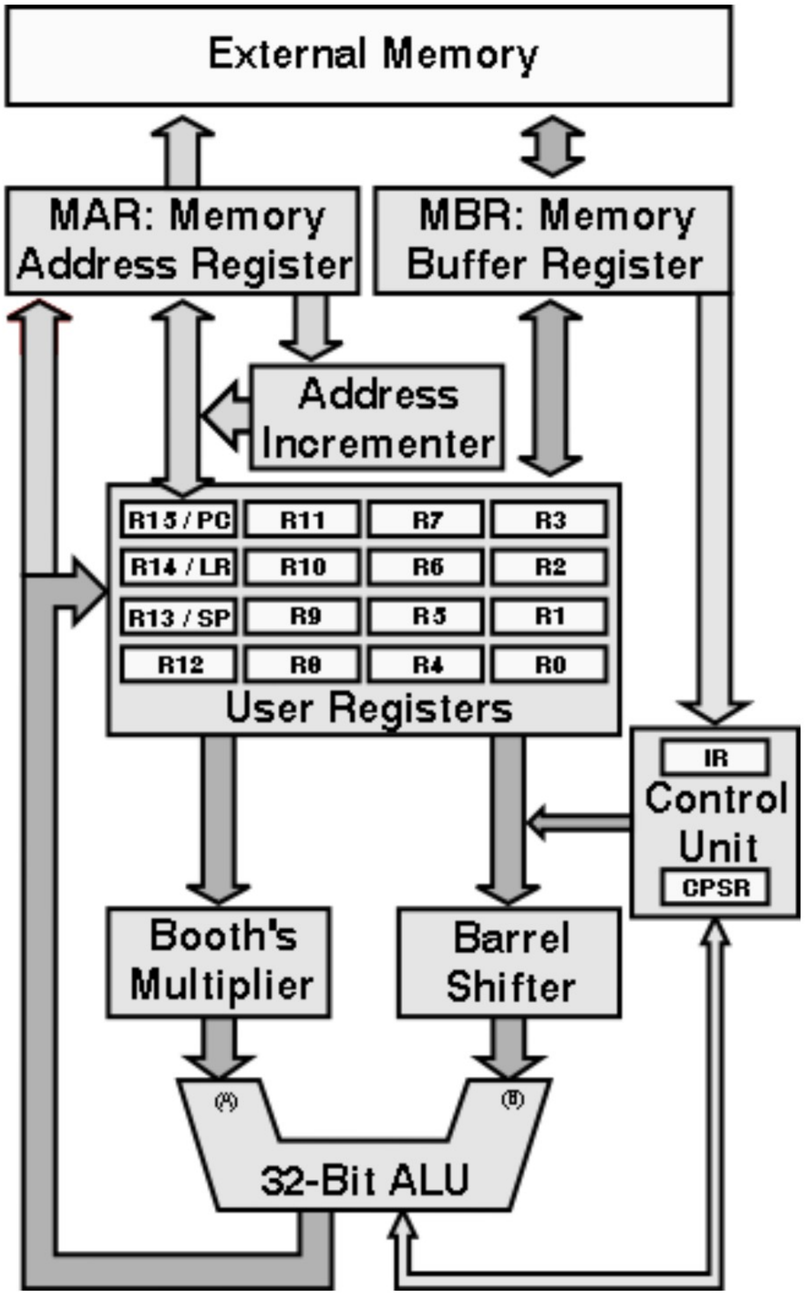
branches to LOOP if [R5] = [R6]

Conditional branch: `BEQ R5, R6, LOOP`

```
1. Memory address ← PC,
   Read memory,
   IR ← Memory data,
   PC ← PC + 4

2. Decode instruction,
   RA ← R5,
   RB ← R6

3. RZ ← RA - RB,
   If $ALU_{iszero}$ = 1 then PC ← PC + Branch offset

4. No action

5. No action
```

ALU has flags for zero, positive, negative, overflow, carry out

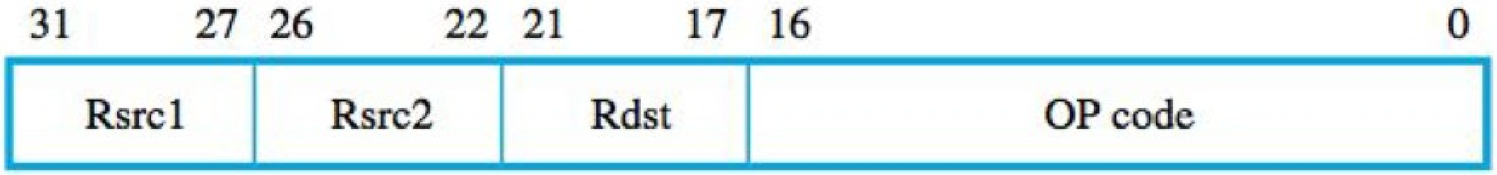
ARM Datapath



Control design

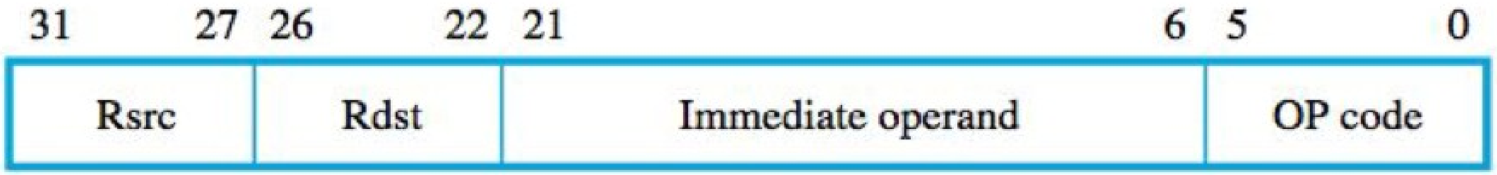
Textbook 5.5-5.7

Example RISC instruction format



31 – 27	26 – 22	21 – 17	16 – 0
Rsrc1	Rsrc2	Rdst	OP code

(a) Register-operand format

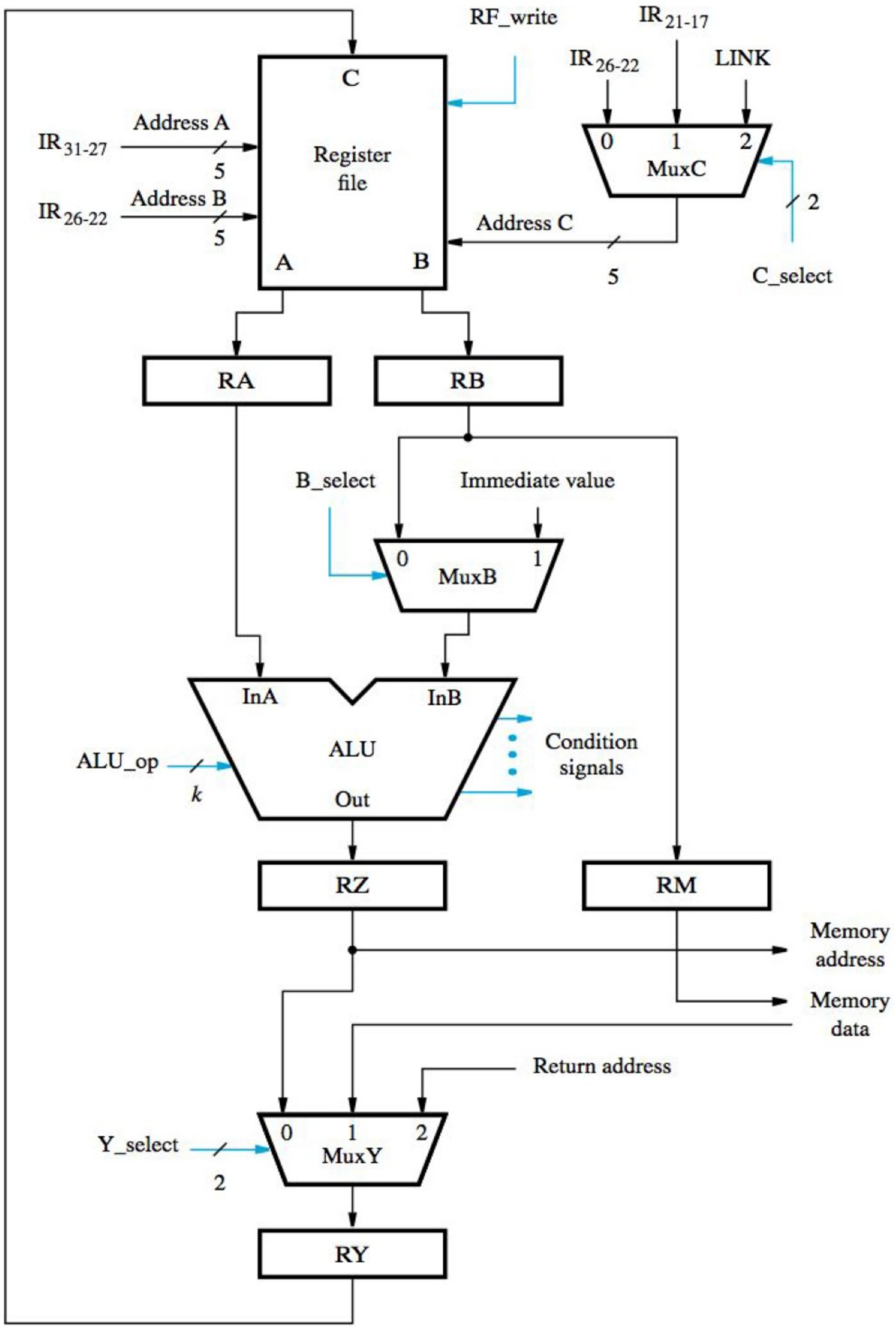


31 – 27	26 – 22	21 – 6	5 – 0
Rsrc	Rdst	Immediate operand	OP code

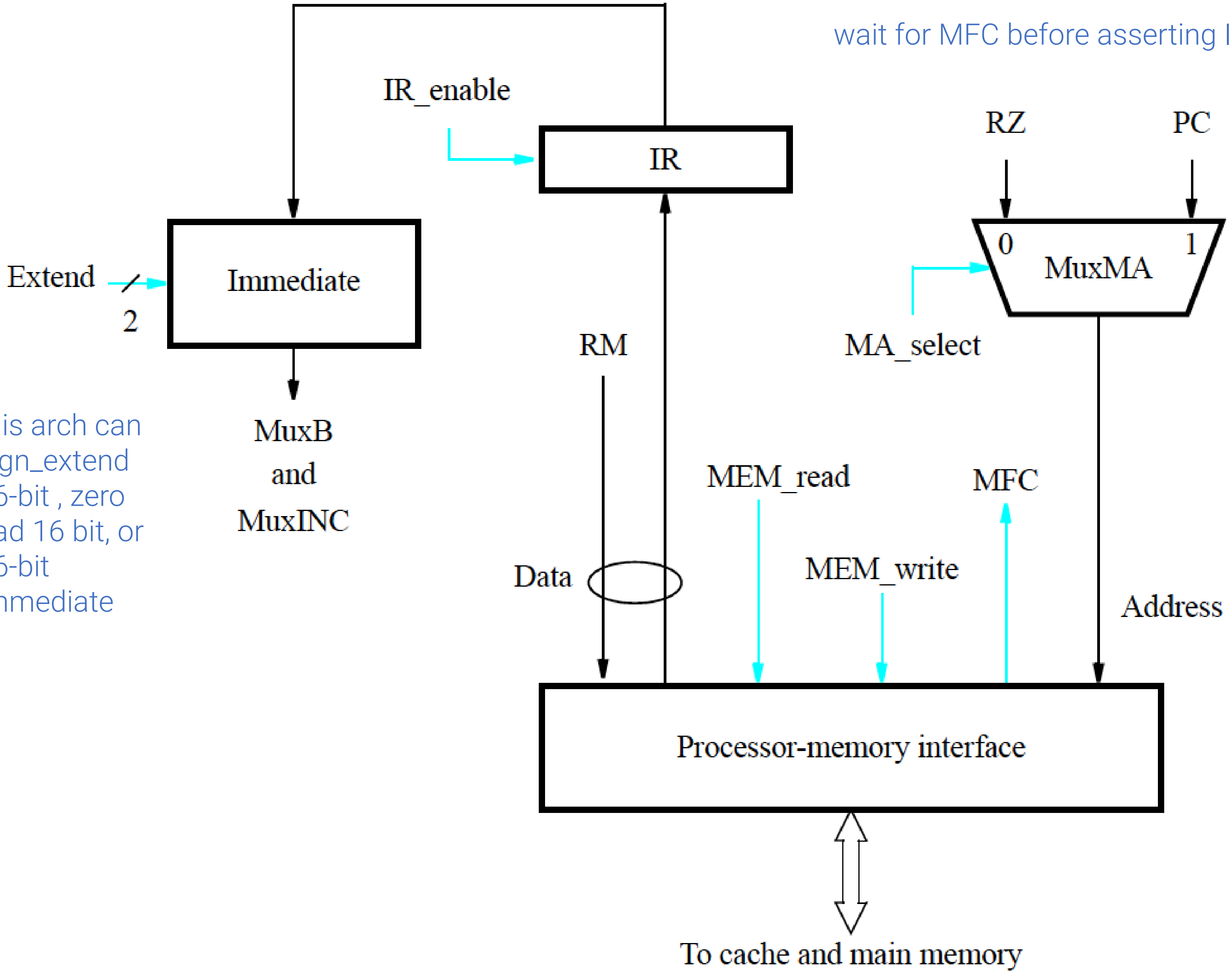
(b) Immediate-operand format

Control signals

- Select multiplexer inputs to guide the flow of data.
- Set the function performed by the ALU.
- Determine when data are written into the PC, the IR,
  the register file, and the memory.
- Inter-stage registers are always enabled because
  their contents are only relevant in the cycles for which
  the stages connected to the register outputs are active.



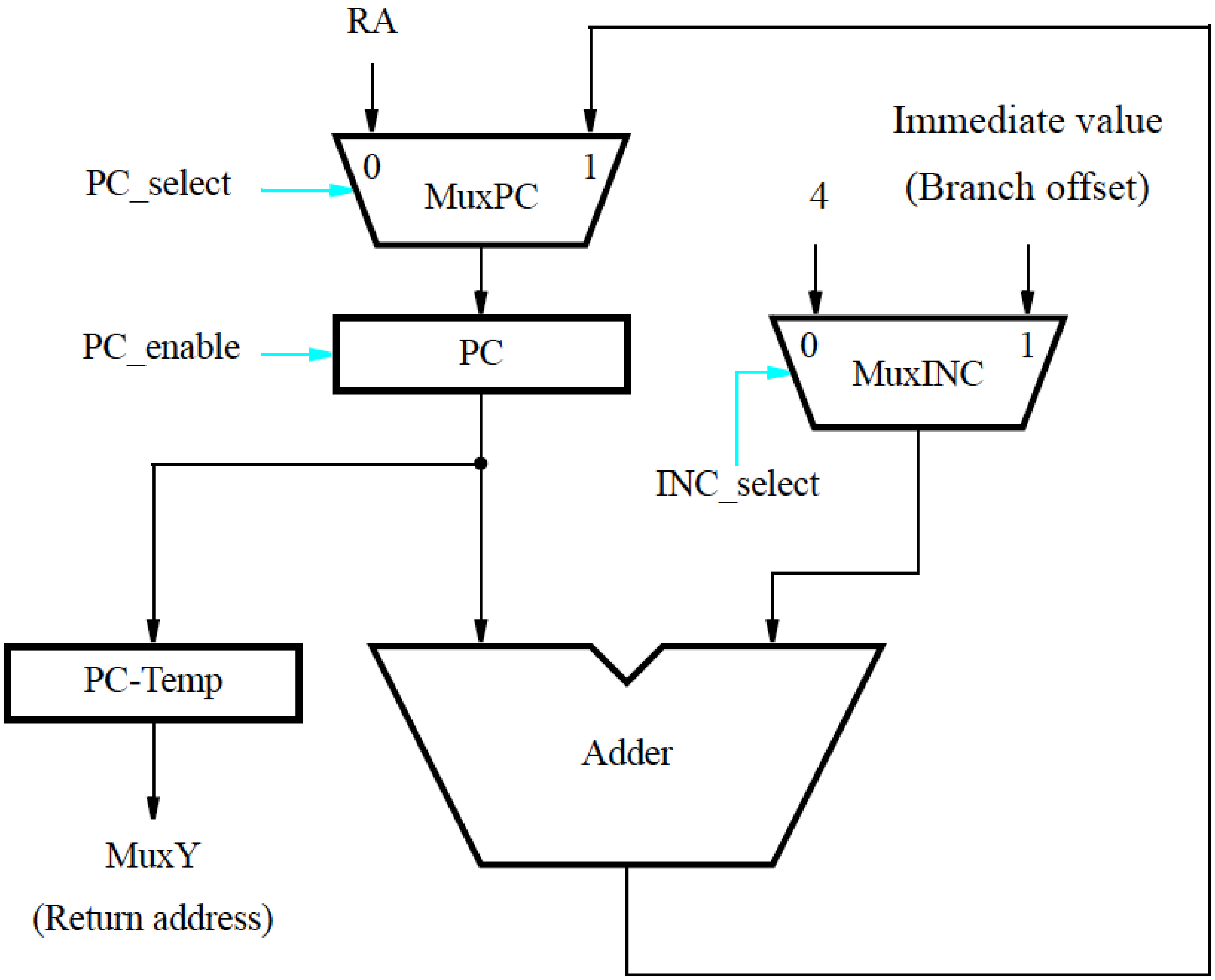
Memory and IR control signals



wait for MFC before asserting IR_enable

this arch can sign_extend 16-bit , zero pad 16 bit, or 26-bit immediate

Control signals of instruction address generator



Control signal generation

- The control unit generates the control signals so the actions in the datapath take place in the correct sequence and at the correct time.
- Two basic approaches:
  1. hardwired control
  2. microprogramming
- *Hardwired control* involves implementing a finite state machine (FSM).
- The state of the FSM is kept in a counter that keeps track of the execution step (one clock cycle / each of the 5 steps, unless a memory access takes more than one cycle)
- The inputs to the FSM are the IR, ALU result (result of a computation, comparison), and external inputs such as interrupt requests
- The outputs are the control signals

Hardwired generation of control signals

- Example: step 1 (fetch)

T1 = 1

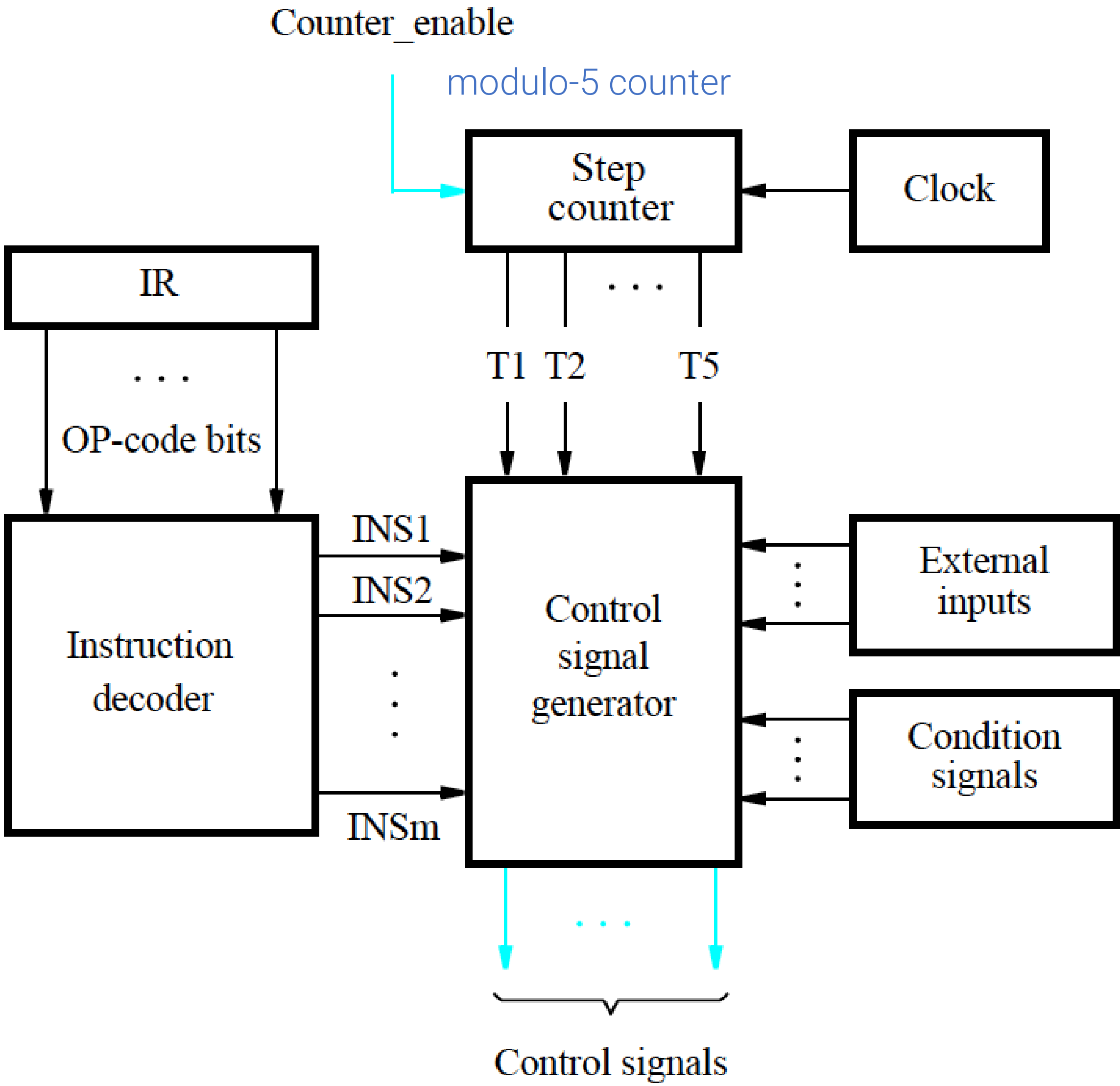
MA_select = 1

MEM_read = 1

IR_enable = 1 when MFC asserted

PC incremented by 4 by setting INC_select to 0 and PC_select to 1

PC_enable = 1



Example

- Wait until **`MFC`** to be asserted before incrementing step counter in a step in which **`MEM_read`** or **`MEM_write`** command is issued
- **`Counter_enable`** should be set to 1 in any step in which **`WFMC`** (wait for memory complete) is not asserted otherwise, it should be set to one when **`MFC`** is asserted
- What is the logic expression for the **`Counter_enable`** signal?

$$\mathrm{Counter}_{\mathrm{enable}} = \overline{\mathrm{WMFC}} + \mathrm{MFC}$$

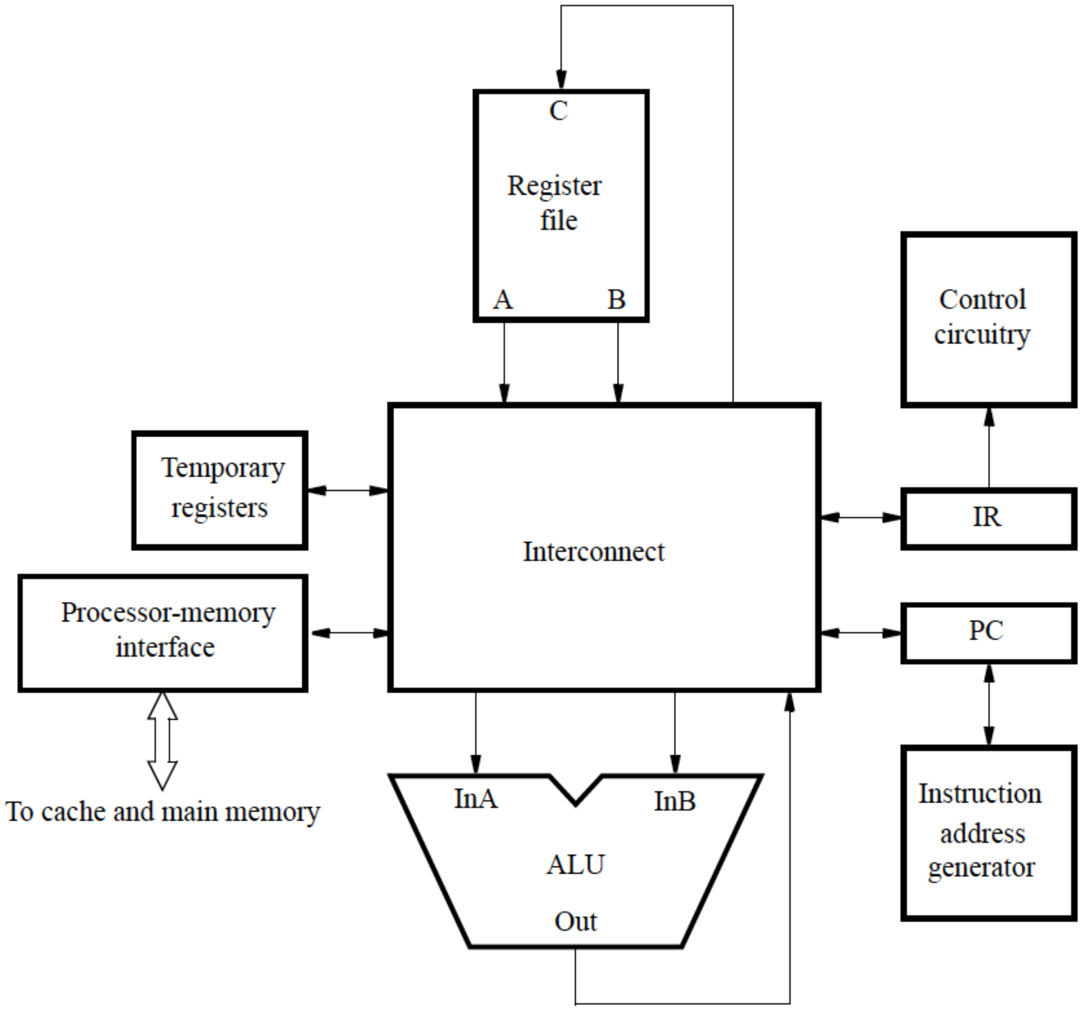
Example

- Make sure the `PC` is incremented only once when a execution step is extended for more than one clock cycle
- `PC` should only be enabled when `MFC` is asserted, also in step 3 of branch instructions
- What is the logic expression for the `PC_enable` signal?

$$\mathrm{PC}_{\mathrm{enable}} = \mathrm{T1} \cdot \mathrm{MFC} + \mathrm{T3} \cdot \mathrm{branch}$$

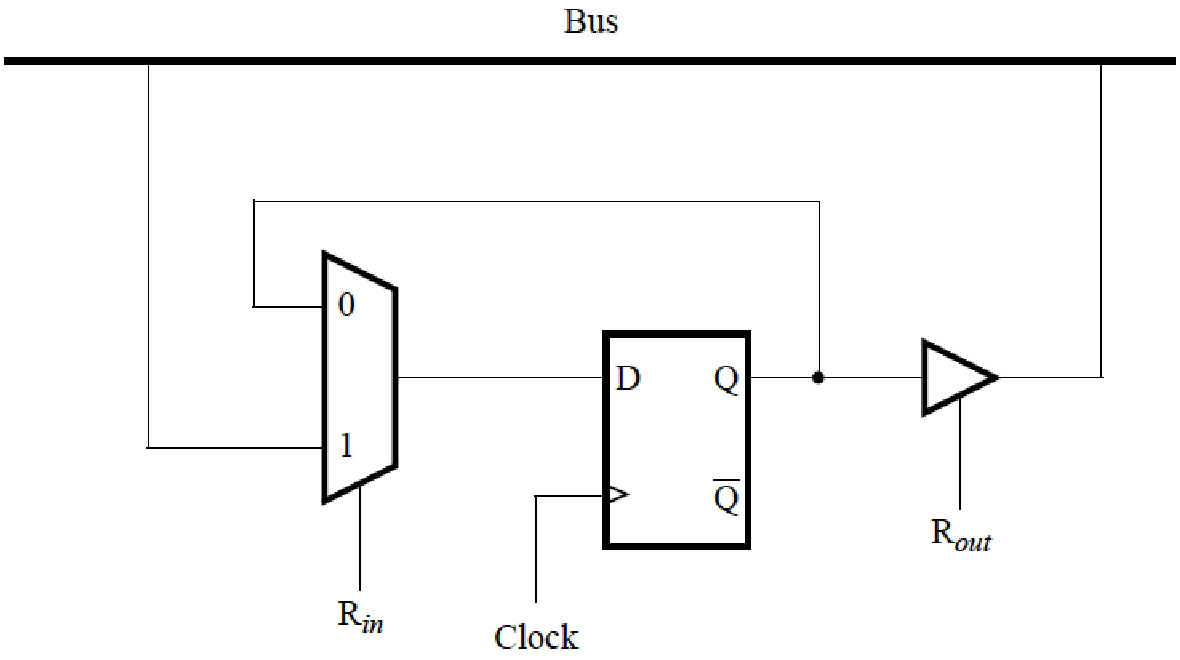
CISC processors

- CISC-style processors have more complex instructions.
- Addressing modes that allow operands to be in memory, variable-length instructions



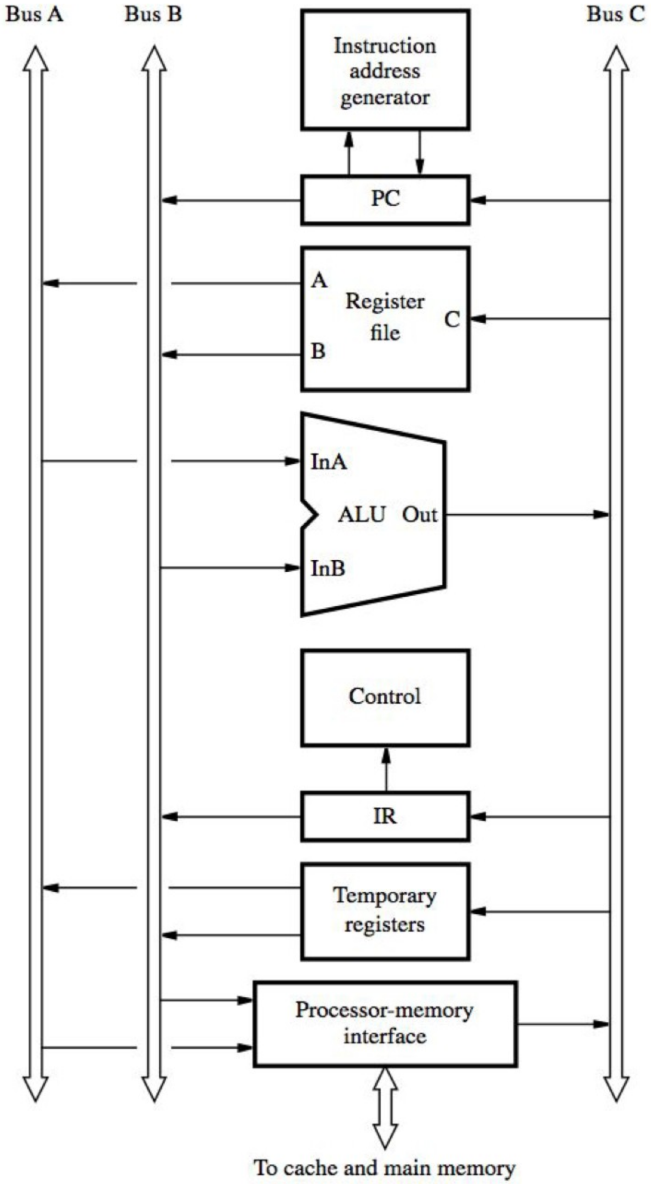
Bus

- An example of an interconnection network.
- When functional units are connected to a common bus, tri-state drivers are needed.



A 3-bus CISC organization

addresses and immediate connections not shown



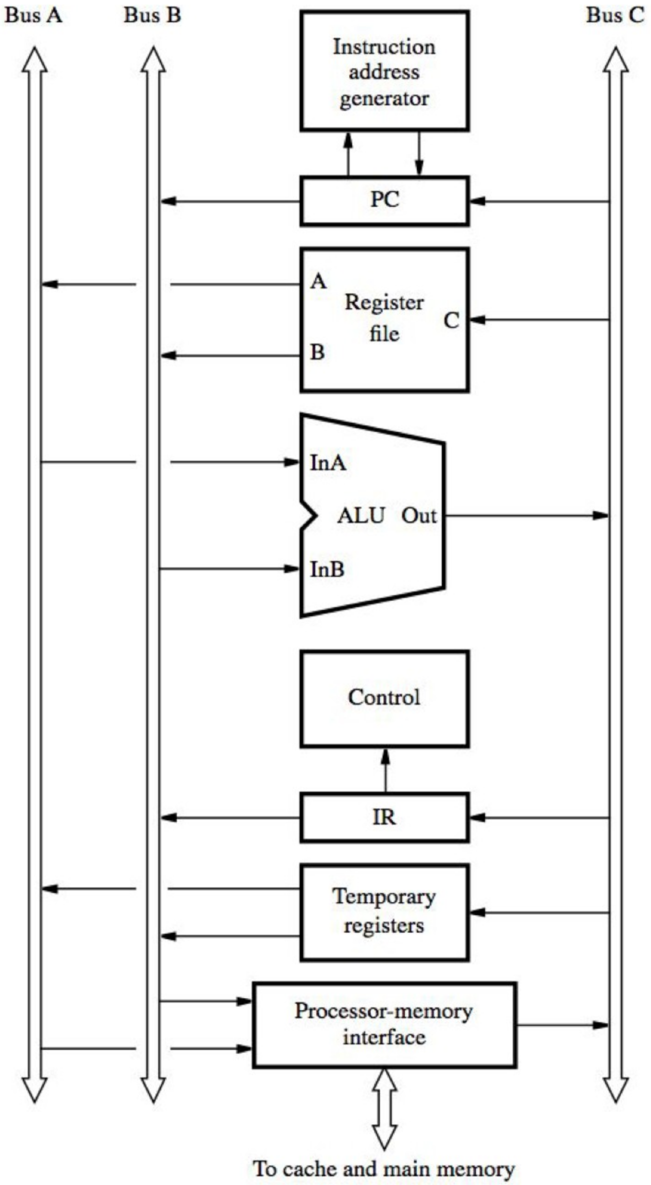
Example: `AND X(R7), R9` (2 word instruction)

fetch opcode

```
1. Memory address ← PC,
   Read memory, Wait for MFC,
   IR ← Memory data, PC ← PC + 4

2. Decode instruction
```

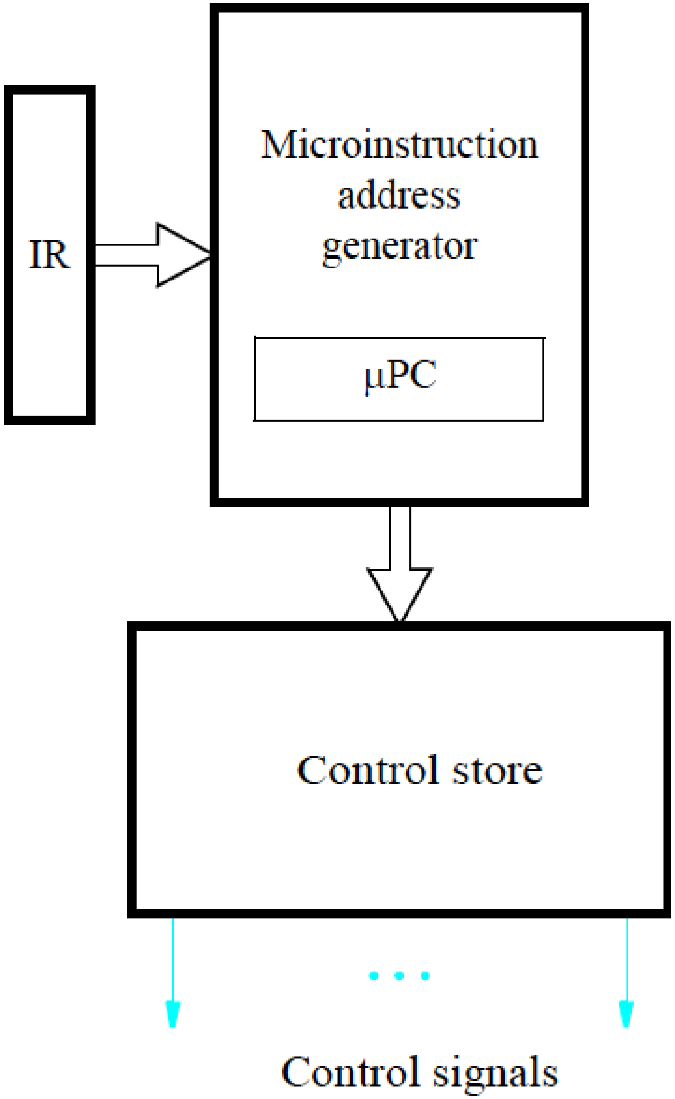
fetch second instruction word (X)

```
3. Memory address ← PC,
   Read memory, Wait for MFC,
   Temp1 ← Memory data, PC ← PC + 4

4. Temp2 ← Temp1 + R7

5. Memory address ← Temp2,
   Read memory, Wait for MFC,
   Temp1 ← Memory data

6. Temp1 ← Temp1 AND R9

7. Memory address ← Temp2,
   Memory data ← Temp1,
   Write memory, Wait for MFC
```



Microprogrammed control

- *Microprogramming* is a software-based approach for the generation of control signals.
- The values of the control signals for each clock period are stored in a *microinstruction* (control word) in a special memory.
- A processor instruction is implemented by a sequence of microinstructions.
- From decoding of an instruction in `IR`, the control circuitry executes the corresponding sequence of microinstructions.
- μPC maintains the location of the current microinstruction.



IR

Microinstruction address generator

μPC

Control store

. . .

Control signals

Microprogramming

- Microprogramming provides the flexibility needed to implement more complex instructions in CISC processors.

- However, reading and executing microinstructions incurs undesirably long delays in high-performance processors.

Pipelining

Textbook 6.1-6.7

Example

3 hours to complete one load

Wash

Dry

Fold

Example

6 hours to complete two loads

Wash

Dry

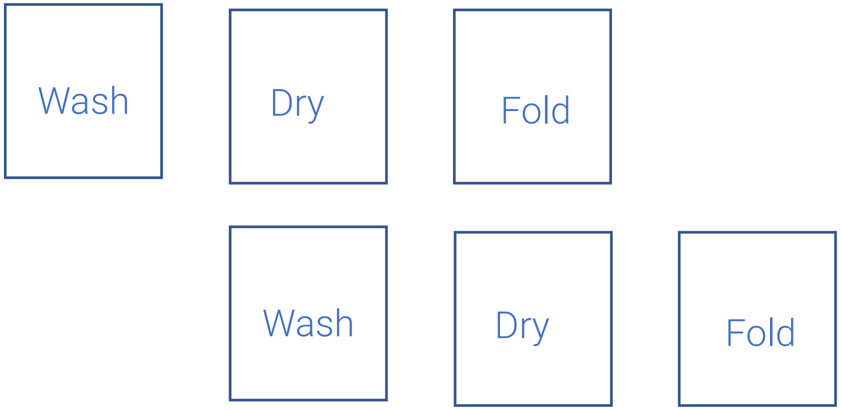
Fold

Wash

Dry

Fold

Example

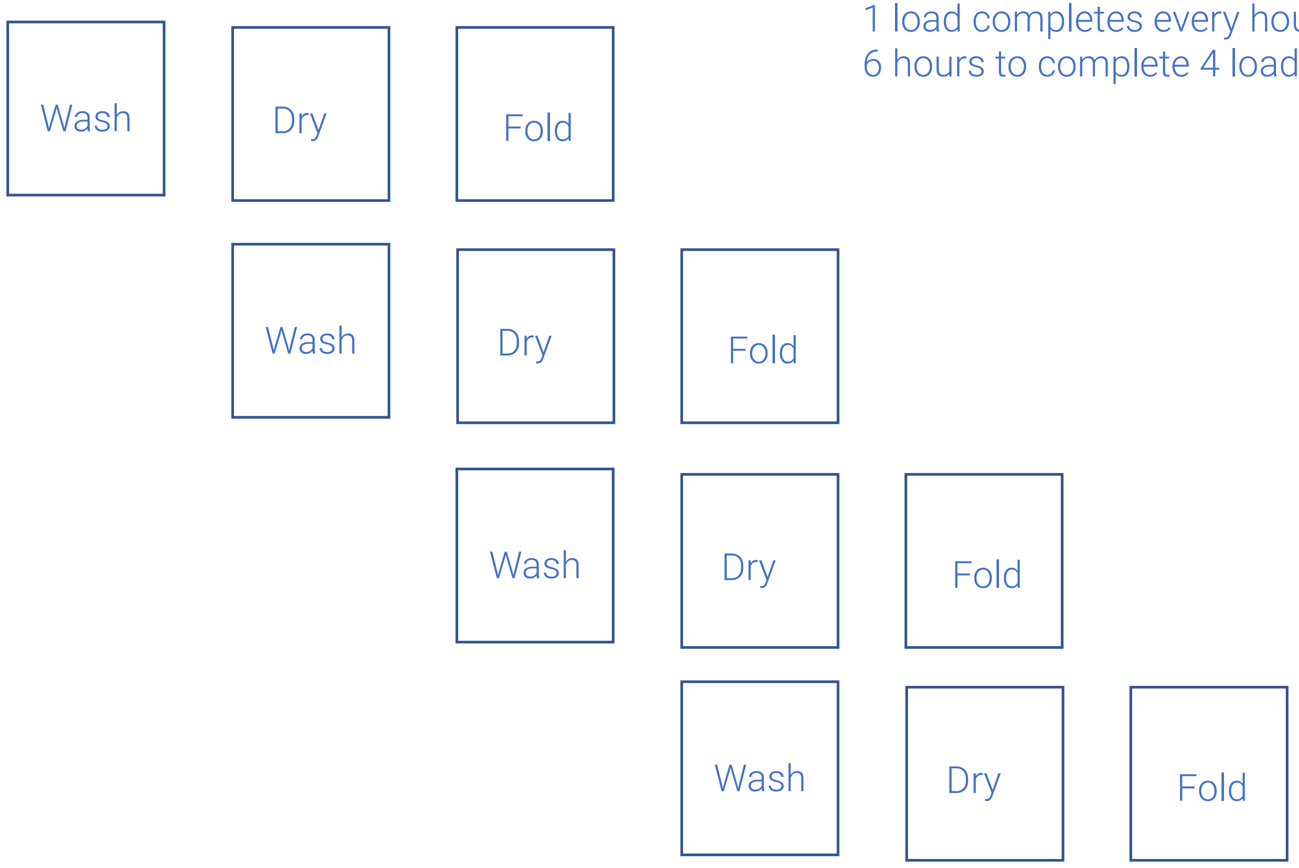


Wash Dry Fold

Wash Dry Fold

Example

3 hours to complete one load
1 load completes every hour
6 hours to complete 4 loads



Wash Dry Fold

Wash Dry Fold

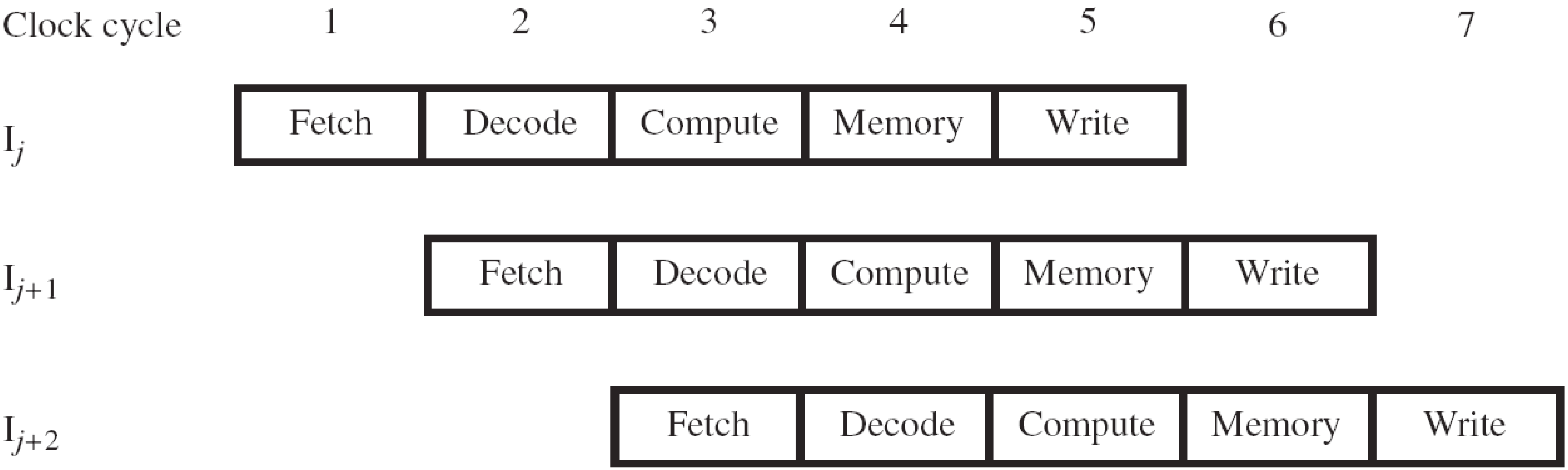
Wash Dry Fold

Wash Dry Fold

What is pipelining?

- Pipelining is applying the ”assembly line” concept to the execution of instructions
- Multiple instructions can be executed simultaneously !
- Divide the instruction into distinct steps (e.g. 5 steps)
- Overlap the execution of the five steps by allowing the hardware for each step to work on the next instruction in the program after it is finished working on the current instruction

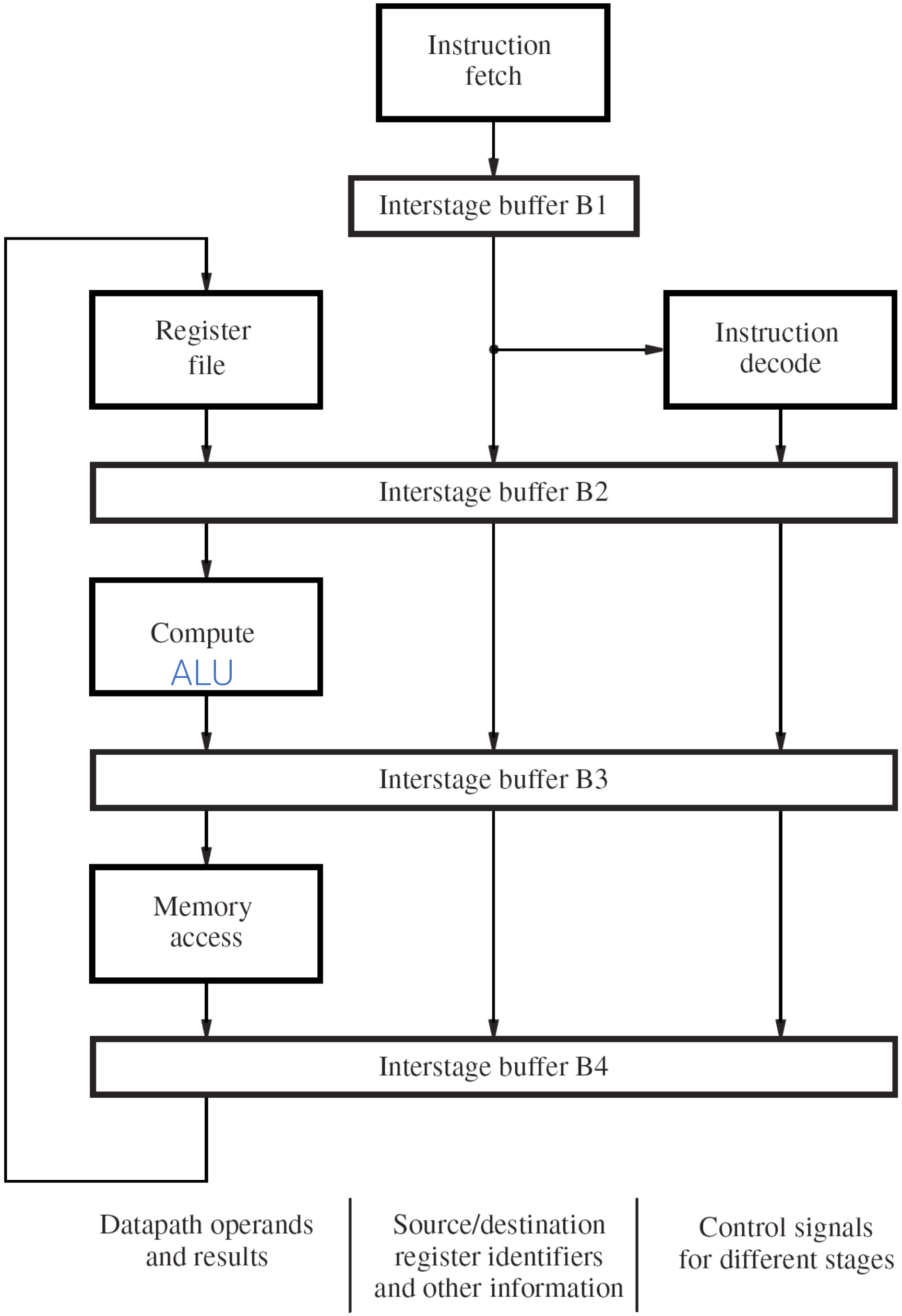
Each instruction still takes 5 cycles to execute, but the rate is 1 instruction per cycle



Clock cycle	1	2	3	4	5	6	7
I_j	Fetch	Decode	Compute	Memory	Write		
I_{j+1}		Fetch	Decode	Compute	Memory	Write	
I_{j+2}			Fetch	Decode	Compute	Memory	Write

Pipeline Organization

- Use program counter (`PC`) to fetch instructions
- A new instruction enters pipeline every cycle
- Carry along instruction-specific information as instructions flow through the different stages
- Use *interstage buffers* or *pipeline registers* to hold this information
- These buffers incorporate `RA, RB, RM, RY, RZ, IR`, and `PC-Temp` registers
- The buffers also hold control signal settings



Instruction fetch

Interstage buffer B1

Register file

Instruction decode

Interstage buffer B2

Compute

ALU

Interstage buffer B3

Memory access

Interstage buffer B4

Datapath operands and results

Source/destination register identifiers and other information

Control signals for different stages

Data Dependencies

```
ADD      R2, R3, R7
SUB      R9, R2, R8
```

- Destination `R2` of Add is a source for Subtract

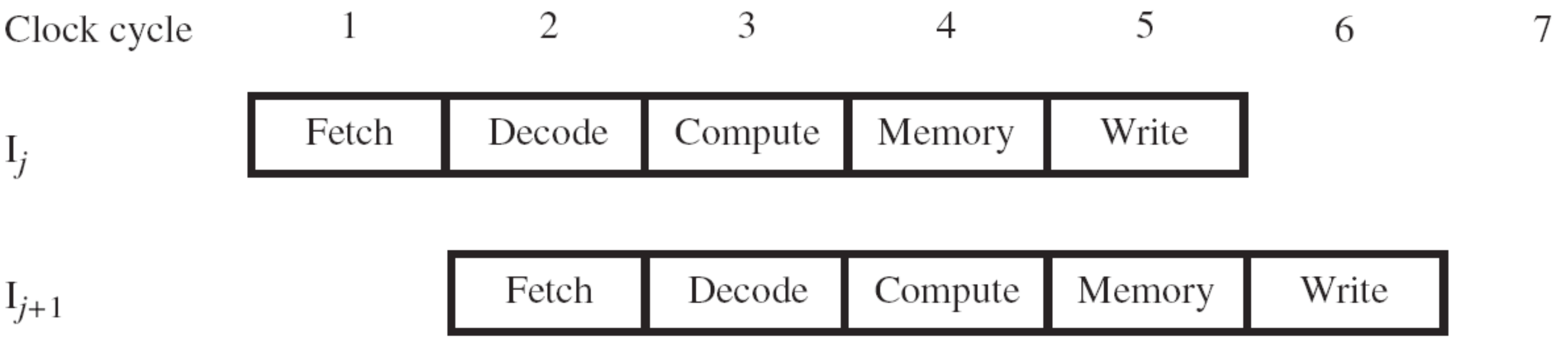
- There is a data dependency between them because `R2` carries data from Add to Subtract

- On *non*-pipelined datapath, result is available in `R2` because Add completes before Subtract

I_j: ADD R2, R3, R7

I_{j+1}: SUB R9, R2, R8

result of I_j written



Clock cycle	1	2	3	4	5	6	7
I_j	Fetch	Decode	Compute	Memory	Write		
I_{j+1}		Fetch	Decode	Compute	Memory	Write	

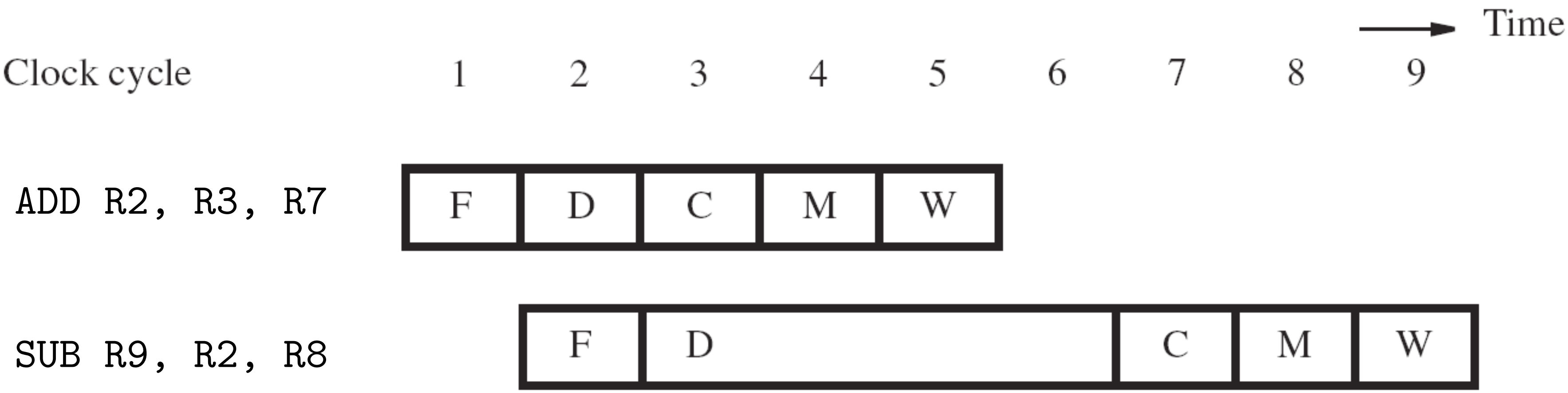
I_{j+1} reads old value of R1 !
→ incorrect result

- The pipeline does not allow the simultaneous execution of these particular instructions because of the data dependency

- This is called a *data hazard*

Stalling the Pipeline

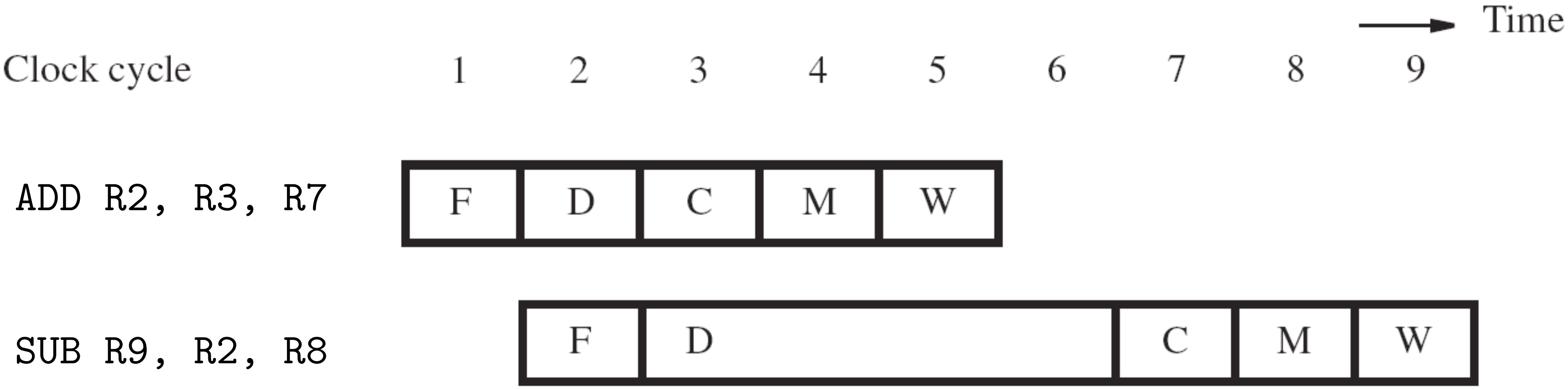
- With pipelined execution, old value is still in register **`R2`** when Subtract is in Decode stage
- So *stall* Subtract for 3 cycles in Decode stage
- New value of R2 is then available in cycle 6



Clock cycle	1	2	3	4	5	6	7	8	9
`ADD R2, R3, R7`	F	D	C	M	W				
`SUB R9, R2, R8`		F	D				C	M	W

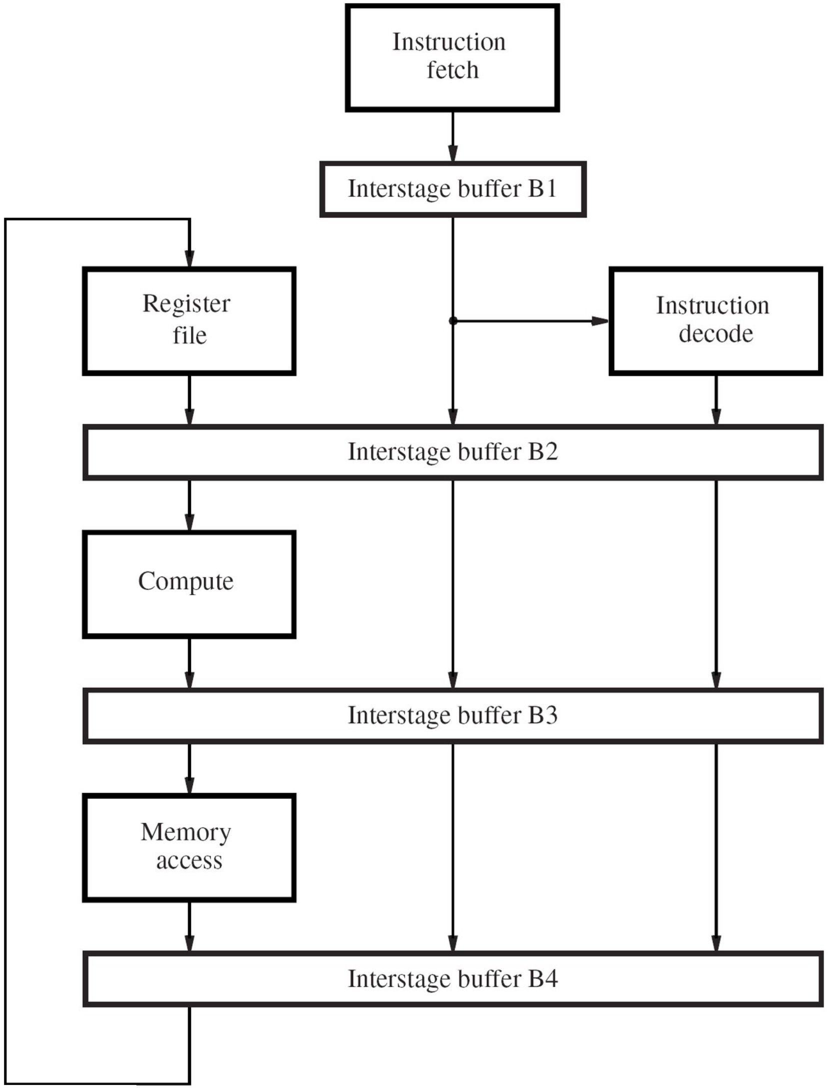
Details for Stalling the Pipeline

- Control circuitry must recognize dependency while Subtract is being decoded in cycle 3
- Interstage buffers carry register identifiers for source(s) and destination of instructions
- In cycle 3, compare destination identifier in Compute stage against source(s) in Decode
- R2 matches, so Subtract kept in Decode while Add allowed to continue normally



Clock cycle	1	2	3	4	5	6	7	8	9
`ADD R2, R3, R7`	F	D	C	M	W				
`SUB R9, R2, R8`		F	D				C	M	W

- Stall the Subtract instruction for 3 cycles by holding interstage buffer B1 contents steady
- But what happens after Add leaves Compute?
- Control signals are set in cycles 3 to 5 to create an *implicit* `NOP` (No-operation) in Compute
- NOP control signals in interstage buffer B2 create a cycle of idle time in each later stage
- The idle time from each NOP is called a *bubble*

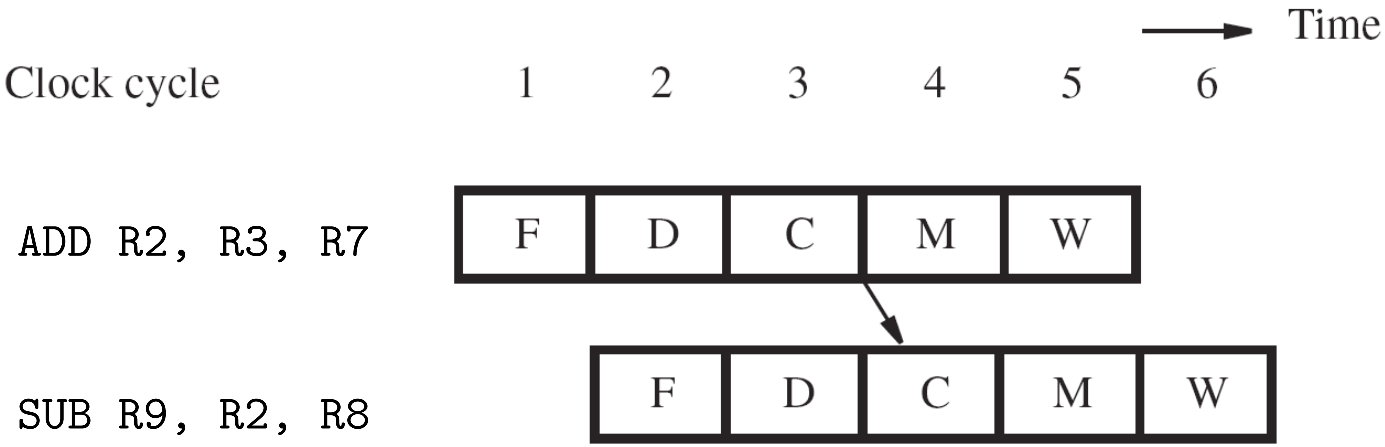


→ Time

Clock cycle	1	2	3	4	5	6	7	8	9
`ADD R2, R3, R7`	F	D	C	M	W				
`SUB R9, R2, R8`		F	D				C	M	W

Can we avoid stalls?

- *Operand forwarding* handles some dependencies without the penalty of stalling the pipeline
- For the preceding sequence of instructions, new value for R2 is available at end of cycle 3
- *Forward* value to where it is needed in cycle 4

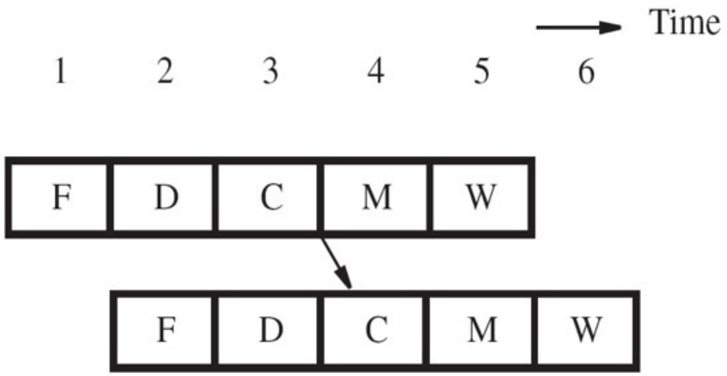


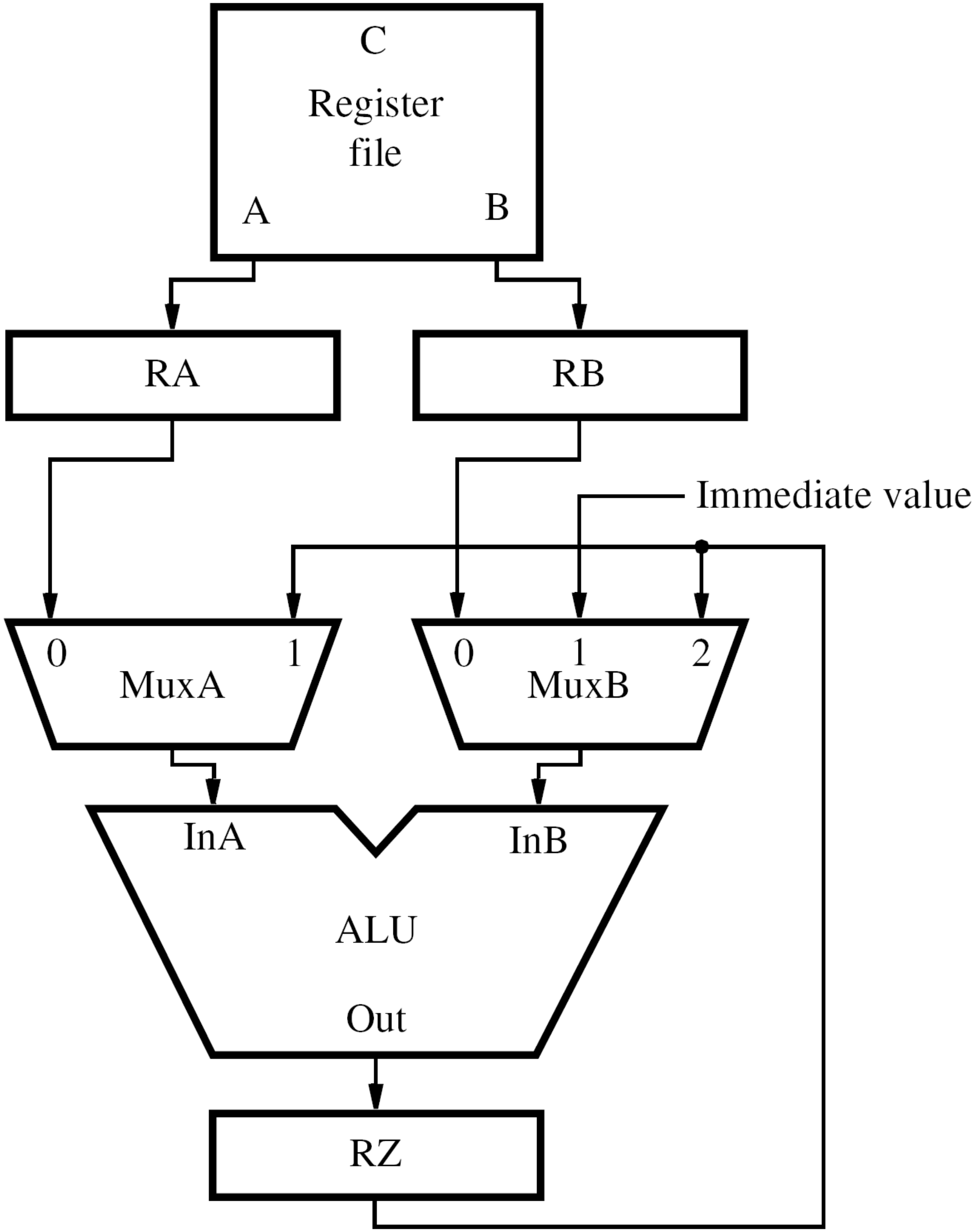
Time

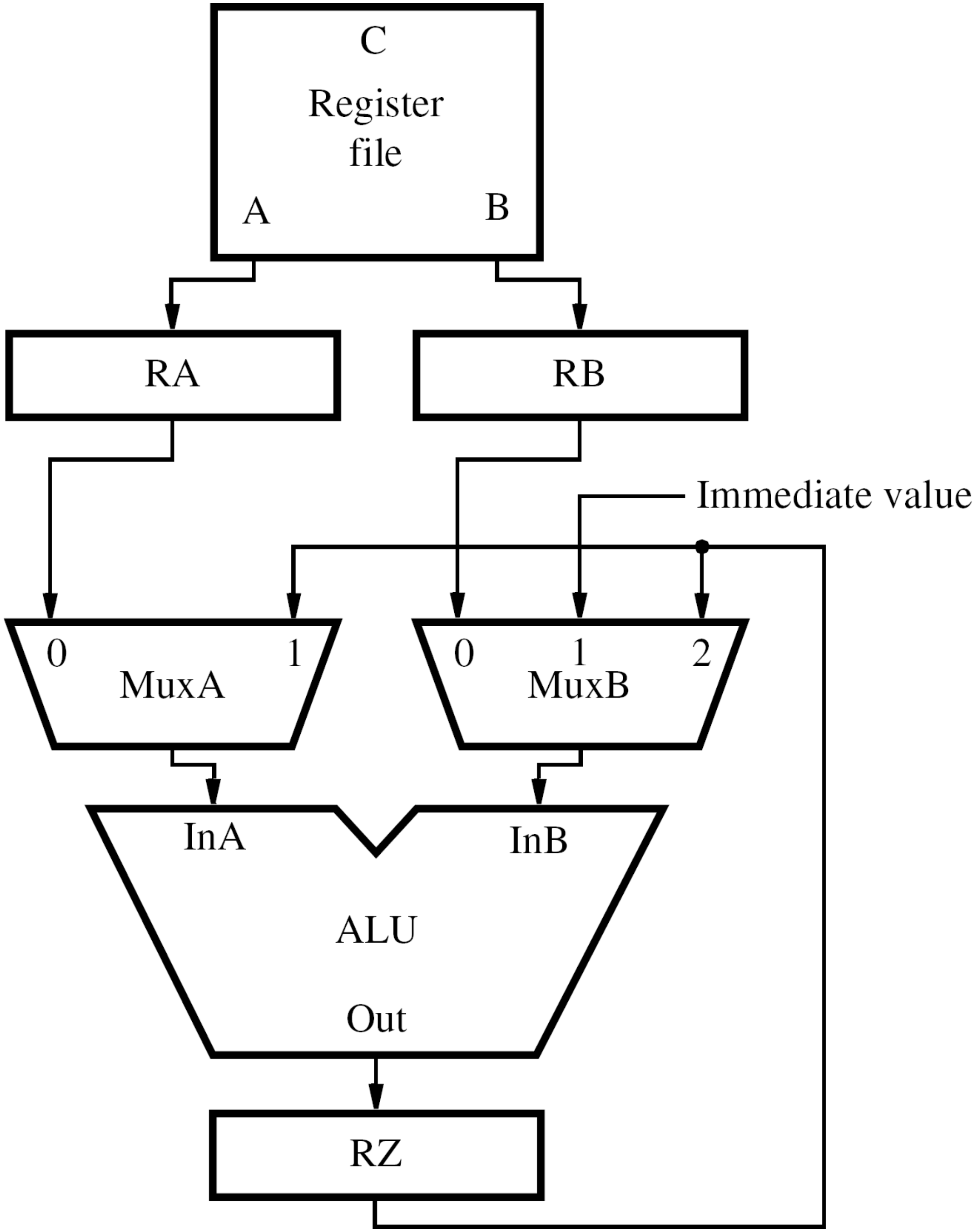
Clock cycle	1	2	3	4	5	6
`ADD R2, R3, R7`	F	D	C	M	W	
`SUB R9, R2, R8`		F	D	C	M	W

Forwarding hardware

Introduce multiplexers before ALU inputs to use contents of register `RZ` as forwarded value






C
Register file
A
B
RA
RB
Immediate value
0
1
MuxA
0
1
2
MuxB
InA
InB
ALU
Out
RZ

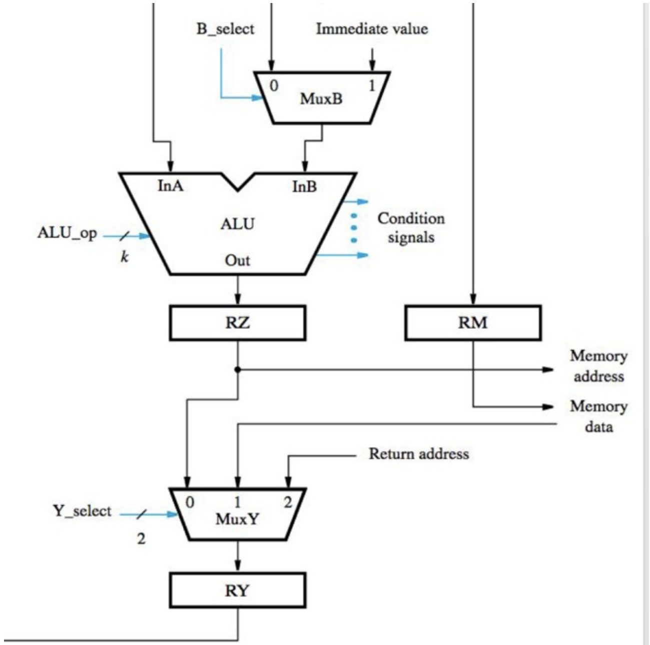
Another example of forwarding

Clock cycle	1	2	3	4	5	6	7
I_j	Fetch	Decode	Compute	Memory	Write		
I_{j+1}		Fetch	Decode	Compute	Memory	Write	
I_{j+2}			Fetch	Decode	Compute	Memory	Write

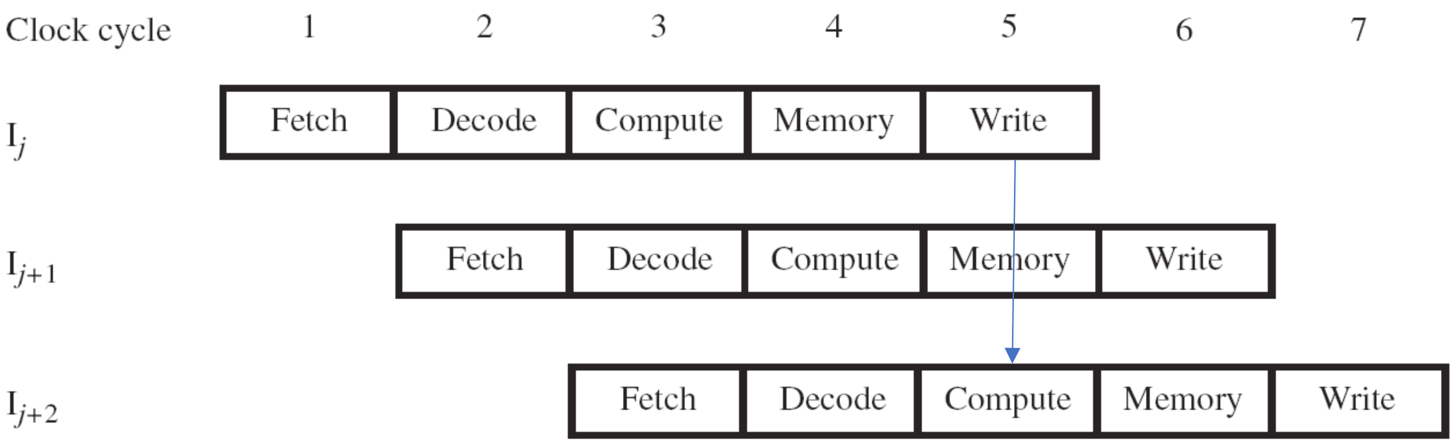
I_j: ADD R2, R3, R7

I_{j+1}: ORR R4, R5, R6

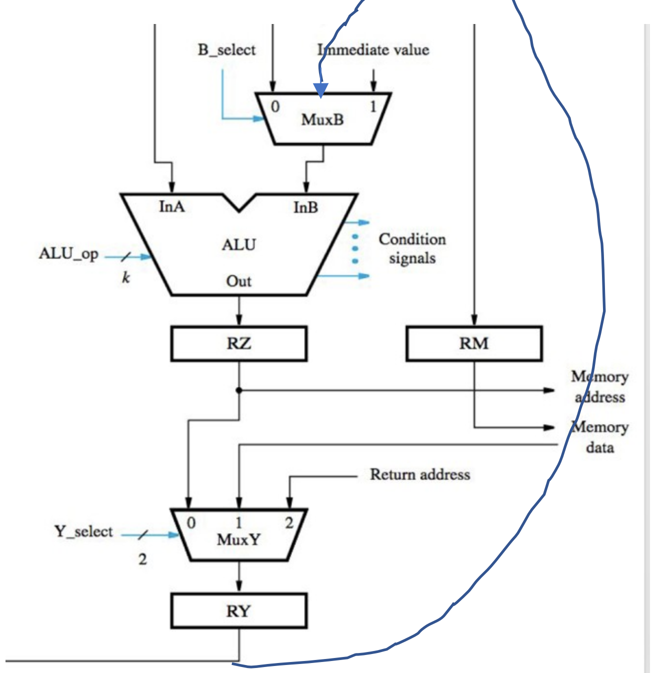
I_{j+2}: SUB R9, R2, R8



Another example of forwarding



Clock cycle	1	2	3	4	5	6	7
I_j	Fetch	Decode	Compute	Memory	Write		
I_{j+1}		Fetch	Decode	Compute	Memory	Write	
I_{j+2}			Fetch	Decode	Compute	Memory	Write



I_j: ADD R2, R3, R7

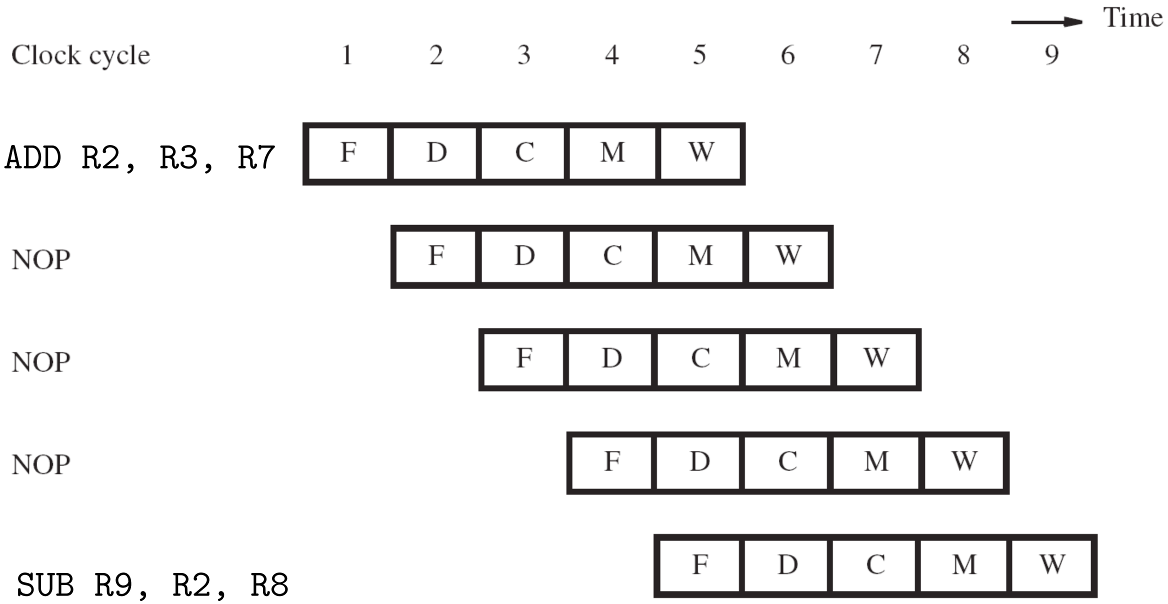
I_{j+1}: ORR R4, R5, R6

I_{j+2}: SUB R9, R2, R8

Extend MuxB to allow forwarded input from RY

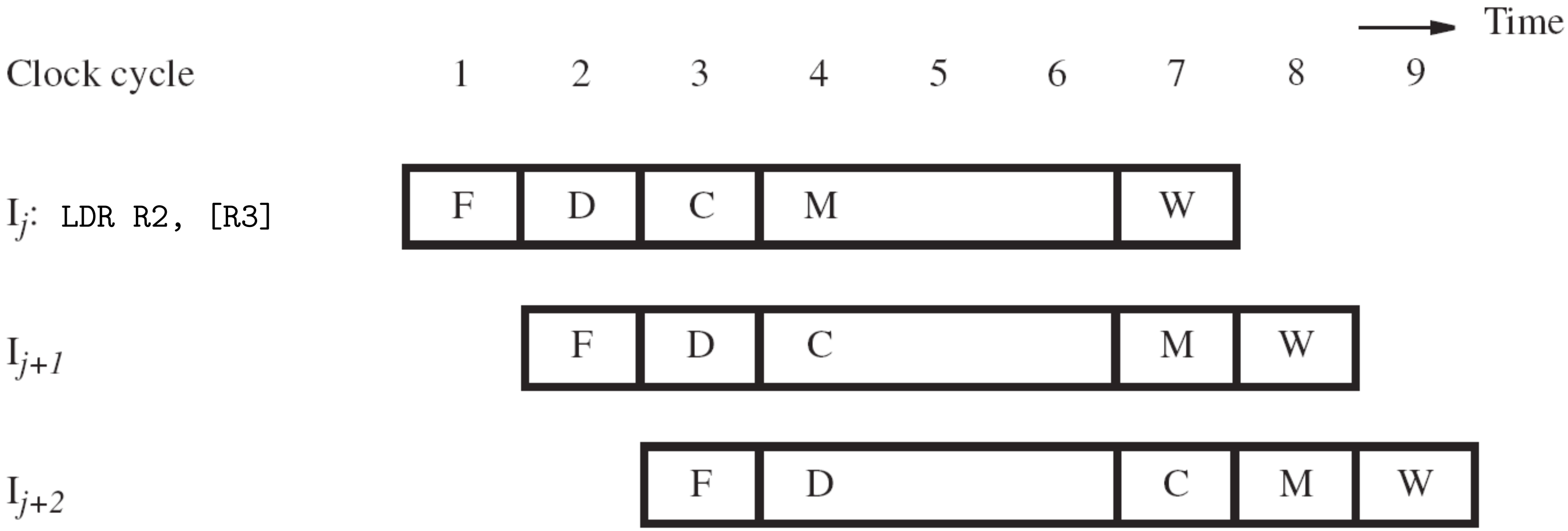
Software Handling of Dependencies

- Data dependencies are evident at compile time
- Compiler puts three *explicit* NOP instructions between instructions having a dependency
- Delay ensures new value available in register but causes total execution time to increase
- Compiler can *optimize* by moving instructions into NOP slots (if data dependencies permit)



Clock cycle	1	2	3	4	5	6	7	8	9
ADD R2, R3, R7	F	D	C	M	W				
NOP		F	D	C	M	W			
NOP			F	D	C	M	W		
NOP				F	D	C	M	W	
SUB R9, R2, R8					F	D	C	M	W

Memory delays



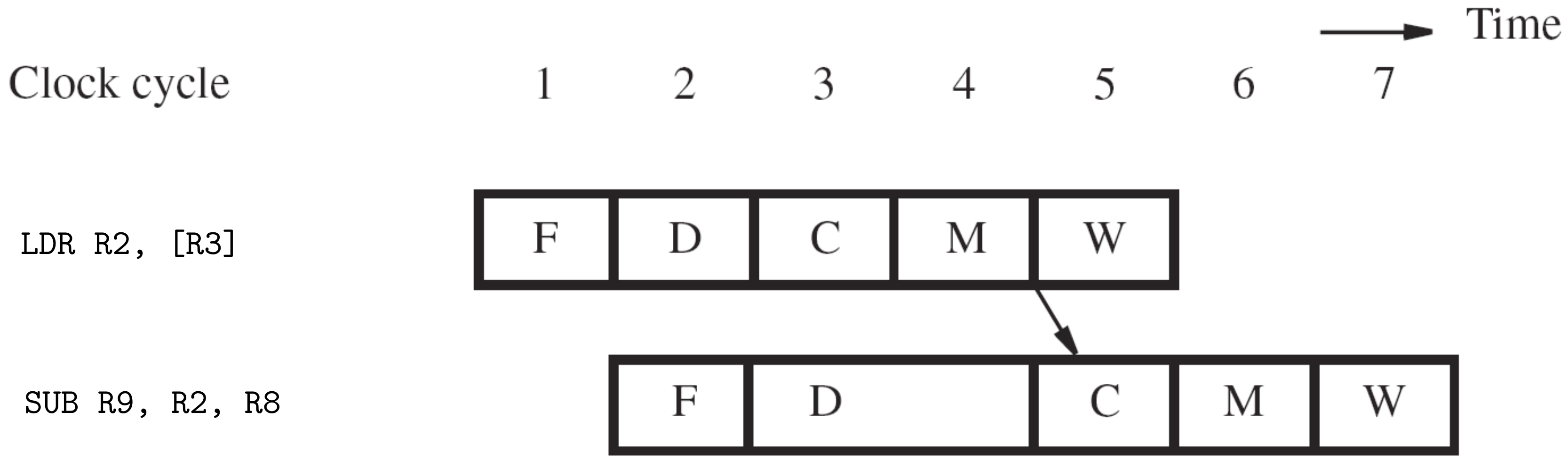
→ Time

Clock cycle	1	2	3	4	5	6	7	8	9
I_j: `LDR R2, [R3]`	F	D	C	M			W		
I_{j+1}		F	D	C			M	W	
I_{j+2}			F	D			C	M	W

Cache misses result in delay in memory stage

Memory Delays

- Even with a cache *hit*, a Load instruction may cause a short delay due to a data dependency
- One-cycle stall required for correct value to be forwarded to instruction needing that value
- Optimize with useful instruction to fill delay



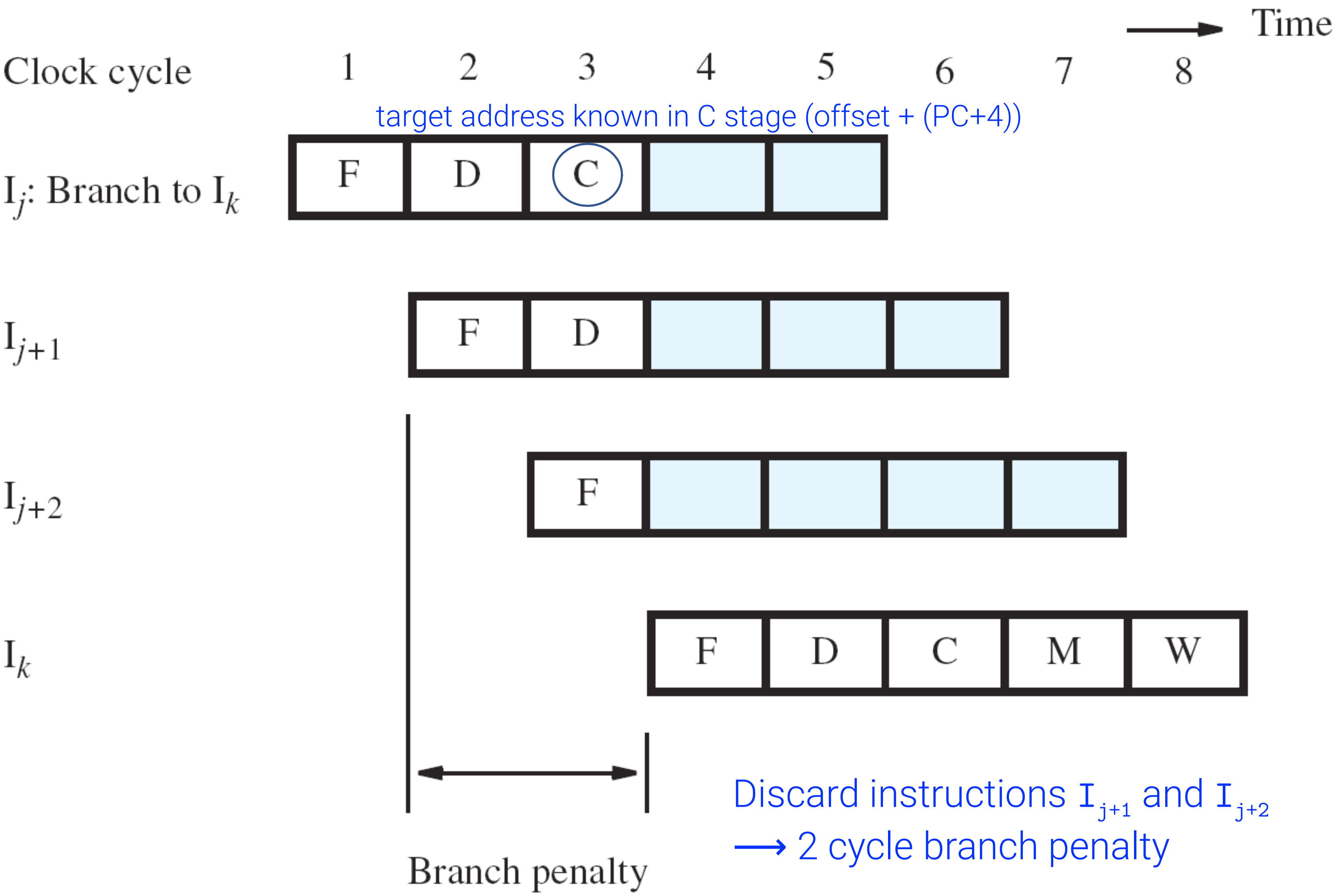
Branch Delays

- Ideal pipelining: fetch each new instruction while previous instruction is being decoded

- Branch instructions alter execution sequence – the branch instruction has to compute the target address and also perform a comparison to determine if we go to the target or fall-through instruction next

- Since these computations happen in later clock cycles, there is a hazard created when pipelining a branch instruction

Unconditional Branches

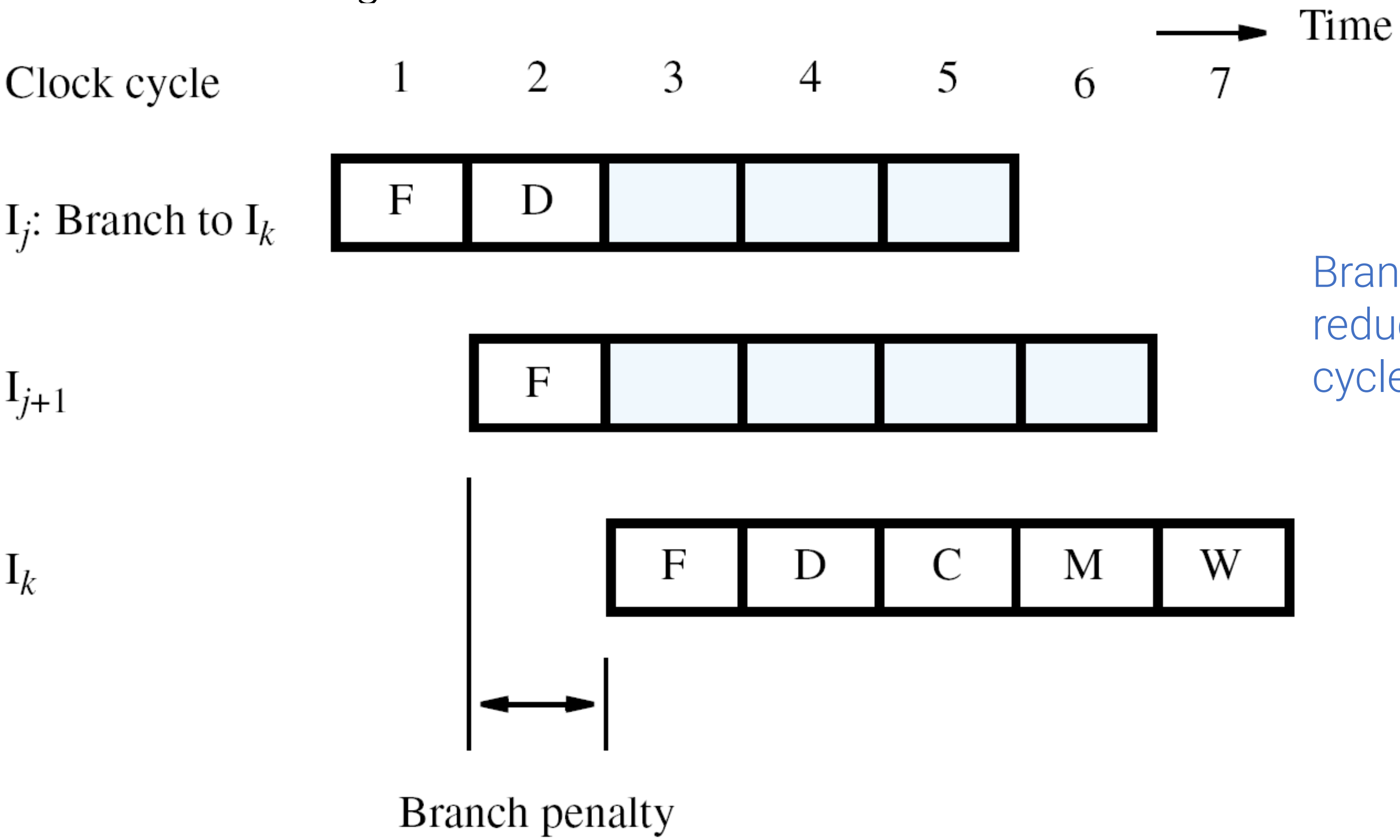


target address known in C stage (offset + (PC+4))

Discard instructions I_{j+1} and I_{j+2}
⟶ 2 cycle branch penalty

Reducing the Branch Penalty

- Must compute the target address earlier in the pipeline
- Introduce a second adder dedicated to computing the branch target in the decode stage



Branch penalty reduced to one cycle

Conditional Branches

```
BEQ  R5, R6, LOOP
```

- Requires not only target address calculation, but also requires comparison in ALU for condition
- Target address now calculated in Decode stage (2 cycle penalty)
- How can we reduce to one-cycle penalty?
  - introduce a comparator just for branches in Decode stage

Delayed branching

- Assuming that both branch decision and target address are determined in Decode stage of pipeline, there is still an unavoidable branch delay of one cycle
- Allow the compiler to fill this *branch delay slot* with a useful instruction, usually from before the branch.
  - Alter the pipeline operation so that the Instruction immediately following a branch is *always* fetched and executed, regardless of branch decision
- If no suitable instruction can be found to fill the slot (because of dependencies) the compiler must put a NOP in the slot

```
                Add                  R7, R8, R9
                Branch_if_[R3]=0     TARGET
                I_{j+1}
                  .
                  .
                  .
TARGET:         I_k
```

(a) Original sequence of instructions containing a conditional branch instruction

```
                Branch_if_[R3]=0     TARGET
                Add                  R7, R8, R9
                I_{j+1}
                  .
                  .
                  .
TARGET:         I_k
```

(b) Placing the Add instruction in the branch delay slot where it is always executed

What's next

- In the next chapter we will look at how to efficiently do arithmetic in computers – how to build fast adders and multipliers for the compute stages in pipelines

- We will also learn about floating-point representation